PRACTICAL SYSTEMS FOR OVERCOMING PROCESSOR IMPERFECTIONS

BY

MATTHEW D. HICKS

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Doctoral Committee:

Associate Professor Samuel T. King, Chair
Professor Sarita Adve
Professor Vikram Adve
Assistant Professor Shobha Vasudevan
Professor David Wagner, University of California at Berkeley

Abstract

Processors are *not* perfect. Even the most modern, thoroughly verified processors contain imperfections. Processor imperfections, being in the lowest layer of the system, pose a significant problem not only for software developers during design and debug, but also serve as weaknesses to the security mechanisms implemented in upper layers. With such a pervasive impact on computing systems, it is vital that processor vendors address these imperfections in a way that maintains the abstraction of a perfect processor promised to software developers.

This thesis proposes SoftPatch, a software-based mechanism for recovering from processor imperfections that preserves the perfect-processor abstraction promised to software developers. By combining the low detection latency of hardware-implemented detectors with lightweight, formally verified software recovery routines, the SoftPatch maintains the illusion of a perfect processor in the face of processor imperfections. SoftPatch uniquely leverages the insights that (1) most of a processor's functionality is thoroughly verified, *i.e.,* free from imperfections, (2) the processor has redundant functionality, and (3) the processor pipeline acts as a checkpointing and rollback mechanism. By leveraging these insights, SoftPatch enables practical patching of processor imperfections. By reusing existing processor features, SoftPatch removes the unneeded complexity and overheads required by previous approaches while still managing to reinforce the perfect-processor abstraction.

To highlight SoftPatch's ability to recover from a range of processor imperfections and to show how systems can be built around SoftPatch, this dissertation presents the design and evaluation of two processor imperfection use cases, processor bugs and malicious processors. We implement detectors for each type of imperfection, one of which we design, and incorporate each use case's detector with SoftPatch into a complete detection and recovery system.

In general, experiments show that SoftPatch is practical and applicable to many sources of processor imperfections. Experiments with the processor bug use case, which we call Erratacator, show that that Erratacator can detect all 16 of the implemented processor bugs and recover from 15. The costs of processor bug recovery are less than 10% hardware area overhead and no run time overhead in the case of monitoring a single bug. Processor bug experiments also show that by exposing the reliability trade-off to software, Erratacator can monitor several processor bugs simultaneously with overheads of less than 10%. Experiments with the malicious processor use case, which we call BlueChip, show that it is able to prevent all implemented hardware attacks, with no incursion on the software developer. Recovery from malicious processor test cases has a small run time overhead and approaching zero hardware area overhead.

To my mom and wife, for their unwavering support.

Acknowledgments

I would not be at the University of Illinois, much less writing this dissertation, if not for Professor Issa Batarseh introducing me to the academic research while an undergraduate at the University of Central Florida. As an undergraduate researcher, my supervisors Dr. Khalid Rustom and Dr. Hussam Alatrash gave me an opportunity to explore an area that would change my lifeFPGAs. Thanks for getting me started.

I want to thank Professor Euripides Montagne for providing extra challenges to me in his classes in an effort to prepare me for a successful graduate student career. I also want to thank him for the extra effort that he gave in personally recommending me to the University of Illinois.

Over the seven years that I have been a graduate student, I have worked with and around many bright and interesting people. I want to thank my many co-workers over the years: Anthony Cozzie, Neal Crago, Nathan Dautenhahn, Murph Finnicum, Haohui Mai, Shuo Tang, and Hui Xue, for the manyseriously manyhours of conversation, spanning topics like research, relationships, politics, and (of course) sports. It was great having a sound- in board for new ideas and pitches and I enjoyed helping them sharpen their ideas.

I also want to thank my research collaborators: Cynthia Sutton, Murph Finnicum, Edgar Pek, and Professors Jonathan Smith and Milo Martin who made doing interesting work and creative ideas possible.

The largest thanks go to my advisor Professor Sam King. Beyond teaching me what research was, how to evaluate it, and how to write about it, he taught me how to be an effective leader; how to keep students expanding their abilities. He also served as a good example of how to attack building a system, inspiring me with a "nothing is too hard" attitude.

Lastly, I thank my committee: Professors Sarita Adve, Vikram Adve, Shobha Vasudevan, and David Wagner for their support and direction.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Software developers must trust the processor. When software developers build a system, they build it upon a set of assumptions. A common assumption that software developers make is that the processor correctly implements the instruction set specification, *i.e.,* they assume the processor is perfect. It is vital that this assumption holds because the processor, being the lowest layer of the system, forms the foundation of any software system. If the processor is incorrect, all functionality that depends on it becomes potentially incorrect.

Contrary to software developers' assumptions, processors are not perfect—they have bugs. Modern processors are as complex as modern software. To create such complexity, processor designers develop processors using a collection of libraries and custom code. Once amassed, the codebase for a modern processor totals several million lines of code [1], which, when fabricated, results in chips comprised of over a billion transistors [2]. Given such complexity, completely verifying modern processors is intractable, meaning each new processor ships with imperfections. For example, Intel's Core 2 Duo processor has an errata document that lists 129 bugs [3]—known bugs. Given the production life of the Core 2 Duo, this amounts to more than 2.5 bugs discovered per month.

Another side effect of processor complexity is an increased risk of *intentional* processor imperfections, referred to as malicious circuits. The millions of lines of code that create a modern processor make an ideal hiding place for malicious circuits. Also adding to the threat of malicious circuits is the trend towards a distributed design model where processor designers respond to demands for increased complexity by outsourcing the implementation of

many components to third parties. This requires that processor designers trust these third parties to be non-malicious. Making concrete the threat of malicious circuits, King *et al.* [4] demonstrate how a malicious processor designer or third party can insert small but powerful attacks into the processor at design time and how an attacker can exploit the attacks at run time to violate software-level security mechanisms.

Existing techniques that could be used to identify malicious circuits have many failings. Code reviews are of little help as it is impractical to expect managers in charge of processor sign-off to visually inspect and understand every line of code. Sign-off managers cannot rely or conventional verification either. Conventional verification fails to find all unintentional imperfections, let alone malicious circuits which are constructed to bypass detection during verification. The lack trusted designers and the limits of functional verification make future processors a prime place for malicious circuits.

Not addressing processor imperfections, malicious or otherwise, breaks the assumption software developers have of a perfect processor. Violating this assumption has several possible consequences, including protracted debugging times, functionality and performance differences [5], and most importantly, security vulnerabilities [6, 7].

1.2 Thesis statement

We propose that we can help software overcome buggy and malicious processors by repurposing functionality already available in the processor.

1.3 The BlueChip recovery system

In this dissertation, we present two systems that aide software in overcoming processor imperfections: one system targeted at helping software overcome errata-like bugs and one system targeted at helping software overcome malicious circuits inserted into the processor during design time. Each system uses hardware-implemented detectors that are responsible for identifying imperfection activations before they cause damage and a software-implemented recovery module that is responsible for helping software execute without ac-

tivating imperfections.

We design the two systems with four key observations in mind:

- *Processor imperfections only matter when they affect the ISA-level [8].* The ISA dictates how software interacts with the processor, so if a fault due to a processor imperfection never makes it to the ISA-level, it will not impact software's execution or any system outputs (ignoring side channels). Thus, the goal of our systems is to remove contaminated state before it reaches the ISA-level.

- *The processor pipeline acts as a checkpointing mechanism for in-flight state.* When an exception occurs, the processor pipeline flushes all intermediate results, creating a window of time where the hardware can view the run time state of the system, but software can not. Combining this observation with the previous observation, we see that if we can detect the activation of processor imperfections while the state that they contaminate is in-flight, we can repurpose the processor pipeline's flush mechanism to maintain a consistent ISA-level state.

- *Most of a processor's functionality is correct: software can depend on it.* Due to the cost of processor bugs, processor designers extensively verify most of a processor's functionality.

- *Processors contain a high degree of functional redundancy.* There are many ways to perform the same task using different instruction sequences. Combining this observation with the previous observation means that there is ample opportunity to recode an instruction stream that activates an imperfection in the processor into a new instruction stream that does not activate the same imperfection.

With those four observations in mind, we propose a generalized system architecture that allows software to overcome buggy and malicious processors. The first component of the architecture is a detector. The detector monitors the hardware-level state of processor, looking for activations of processor imperfections. When it detects an activation, it creates an exception, causing the processor to flush in-flight, potentially contaminated, state. To make sure that no contaminated state makes it to the ISA-level, we require a low-latency detector. Having this, we know that software will never execute in

an inconsistent state. But, this may lock the processor since attempting to execute the same instruction again is likely to re-activate the imperfection.

To help software execute around imperfections, we add a software recovery layer to the system. The recovery layer takes advantage of the functional redundancy in the processor and the fact that most of the processor's functionality is correct to recode instructions from the software stack, routing around the imperfection. The processor, after processing the exception from the detector, passes control to the recovery layer. The recovery layer recodes instructions from the software stack and once execution moves past the imperfection, it returns control back to the software stack.

To address the patching of errata-like processor bugs, we propose Erratacator. Erratacator combines reconfigurable, software-implemented recovery routines with low latency, dynamically configurable, processor bug detectors implemented in hardware to form a unified platform. Erratacator maintains a consistent ISA-level state by connecting previously proposed low-latency processor bug detectors [9] to the processor's pipeline flush mechanism. These detectors are low enough latency that there is *no* need for the heavyweight checkpointing and rollback mechanisms required by software-only bug patching approaches or the micro checkpoints that slow the commit rate used in Diva-like approaches [10].

To eliminate the complex and hardware-intensive recovery requirements of hardware-only processor bug patching approaches, Erratacator employs formally verified, software-level, recovery routines that sit between the processor and the software. The recovery routines act as a middle-man, reading instructions from software, recoding the instruction streams in an effort to route execution around the processor bug(s), and finally returning control back to software.

To address the specific threat of malicious circuits inserted at design time, we present UCI and BlueChip (an early version of the recovery system used in Erratacator). We combine UCI and BlueChip to form a hybrid system, *i.e.,* design-time detection (UCI) and run-time recovery(BlueChip), for detecting and neutralizing potentially malicious circuits. At design time, UCI flags as suspicious, any unused circuitry (any circuit not activated by any of the many design verification tests) and removes their output from the processor, deactivating them. However, these seemingly suspicious circuits might actually perform a legitimate functionality within the processor, so BlueChip

inserts circuitry to raise an exception whenever one of these suspicious circuits would have been activated. BlueChip then takes over as an exception handler and is responsible for simulating software-level instructions in effort to nudge the state of software forward, past the suspicious circuit.

Both of the implemented systems push the complexity of coping with buggy and malicious processors up to a higher, more flexible, and adaptable layer in the system stack, silently maintaining software's assumption of a perfect processor. The key to making this possible with a low software run time overhead is repurposing functionality that ships with the processor.

1.4 Contributions

This section provides a discussion of each major contribution of the work covered in this dissertation. Chapters 4 and 5 include a more extensive list of contributions tailored to the reference design being covered in that chapter.

1.4.1 The first arbitrary malicious circuit identification algorithm targeted at design-time attacks

The computer systems security arms race between attackers and defenders has largely taken place in the domain of software systems, but as hardware complexity and design processes have evolved, novel and potent hardware-based security threats are now possible. We refer to these threats as malicious circuits.

Previous attempts at addressing the threat of malicious circuits focused on supply chain attacks where they assume the existence of a golden version of the hardware [11, 12, 13, 14, 15]. We, instead, look at preventing malicious circuits at design time, where there is no golden reference of how the hardware should behave. Our work is unique from previous research on malicious circuits included at design time because we target arbitrary malicious circuits, while others target specific attack vectors [16].

To identify malicious circuits at design time, we developed UCI. UCI detects suspicious circuits based on the observation that attackers want to ensure that their attacks are not triggered during verification. We evaluated UCI using three malicious circuits (which we also created) that we added to

the Leon3 processor. In less than an hour, UCI was able to detect all three malicious circuits, while identifying less than 1% of the processor circuitry as suspicious.

This work stemmed follow-on work by other researchers [17, 18], focusing more attention on the problem of detecting arbitrary malicious inclusions at design time.

1.4.2 The first to remove suspicious functionality from a processor and use software-level repair to route software execution around the removed functionality

We design an automatic tool to help processor designers cope with the suspicious circuits identified during UCI analysis. This tool removes the threat of suspicious circuits by detaching them from the trusted circuitry. Detaching the suspicious circuits from the trusted circuit makes it impossible for the suspicious circuit to influence the trusted circuit. The problem is that the removed circuits may be part of non-malicious functionality. To protect software from state contamination due to the detached circuitry, our automatic system inserts logic that raises an exception whenever one of the detached circuits would have been activated. The exception handler then becomes responsible for implementing the missing functionality—at the ISA-level of abstraction—that software needs to continue execution.

We evaluate the system using the results from UCI analysis on our three malicious circuits added to the Leon3 processor and three common programs (make, djpeg, and wget) running on Linux. Results show that the system is able to protect the processor from the malicious circuit while at the same time help software make forward progress. Results also show that if the added protections are off the critical timing path for the processor, that both hardware area and power overheads approach zero. In terms of the impact on software, the average run time overhead was less than 1%.

Our contribution here is two-fold. First, we distinguish ourselves from previous work by detaching suspicious circuits. We can detach circuits safely because our recovery software is able to emulate around the removed hardware. In contrast, previous design time defenses used firewalls to limit interactions between untrusted and trusted components [19, 16]. The second contribu-

tion is that our experimental results show that it is possible and practical to push the complexity of coping with malicious hardware up to a higher, more flexible, and adaptable layer in the system stack.

1.4.3 The first implementation and evaluation of a signature-based processor bug detector

In an effort to address the over 70% of modern processor bugs that conventional mechanisms fail to patch [20], previous research proposed using signature-based bug detectors to identify bug activations [9, 21, 20]. The authors of the competing signature-based detectors evaluated their proposals using errata-like bugs to determine, for each bug, how many signals they would have to monitor to detect the bug. The authors stopped short of implementing their detector in hardware and evaluating its run time performance.

Our goal was to learn explore the run time behavior of signature-based detectors, especially the effects of monitoring multiple bugs concurrently. To this end, we implemented 16 bugs in a popular open source processor and ran benchmarks.

Experiments with 9 errata-like bugs from a popular processor show that the detectors can detect all implemented bugs. Running a series of embedded system benchmarks on top of Linux shows that the detectors can monitor a single bug without producing false detections. When we run that same test setup, but try to detect an increasing number of bugs, we show, for the first time, that contention among bugs for the limited bug detector resources creates an overwhelming number of false detections. Specifically, results show that when monitoring the first five bugs, every instruction causes the bug detectors to fire. We also explore the implications of adding more intelligence in determining what bugs are monitored when. By removing a single bug, that we know software will never trigger, from the set of detectable bugs the detectors can monitor all nine bugs with less than 900 false detections per second—a lower frequency than the timer tick on some Linux machines.

Our main contribution here is learning about the scaling difficulty that these types of signature-based detectors have, motivating future work. We also show that software-level recovery is possible on buggy processors.

1.5 Organization

In the next chapter (Chapter 2), we present the concepts necessary for understanding and evaluating the work presented in this dissertation. Chapter 3 covers the work related to protecting software from the implications of buggy and malicious processors. Chapter 3 also details the work related to detecting malicious circuits and design time verification strategies, in general. Chapter 4 covers the design, implementation, and formal verification of the BlueChip recovery firmware and provides an evaluation against errata-like processor bugs. Chapter 5 discusses the design and implementation of our malicious circuit detection and removal tool and a preliminary version of BlueChip that operates as a part of the operating system. We look into possible improvements to the BlueChip recovery system in Chapter 6 and conclude the dissertation in Chapter 7.

Chapter 2

Background

This chapter covers the background topics required for understanding the two reference implementations presented in Chapters 4 and 5.

2.1 Trusted Computing Base

According to Lampson *et al.* [22], the Trusted Computing Base (TCB) for a system is, "A small amount of software and hardware that security depends on and that we distinguish from a much larger amount that can misbehave without affecting security." We build on this definition by refining *security* to mean the security policy of the system. For example, a security policy that an operating system wants to enforce is that untrusted processes can only interact with other processes using the operating system's well-defined interprocess communication abstractions. The TCB for a generalized operating system that implements this security policy consists of the processor and other potentially shared hardware components, the kernel, device drivers, libraries, and trusted processes. These components amount to millions of lines of code—both hardware and software—written by many different entities, with varying levels of verification. With such a large and diverse TCB, it is intractable for system builders to prove that it is trustworthy; which is why systems have security vulnerabilities.

Reducing the size of the TCB is a primary goal of secure system designers. A smaller TCB is easier to reason about or even verify completely. One way to reduce TCB size is by moving components outside of the TCB, *i.e.,* make them untrusted. Another way to reduce the size of the TCB is to reduce the complexity of the trusted components; reducing complexity often yields more functionally correct and more readily verifiable components.

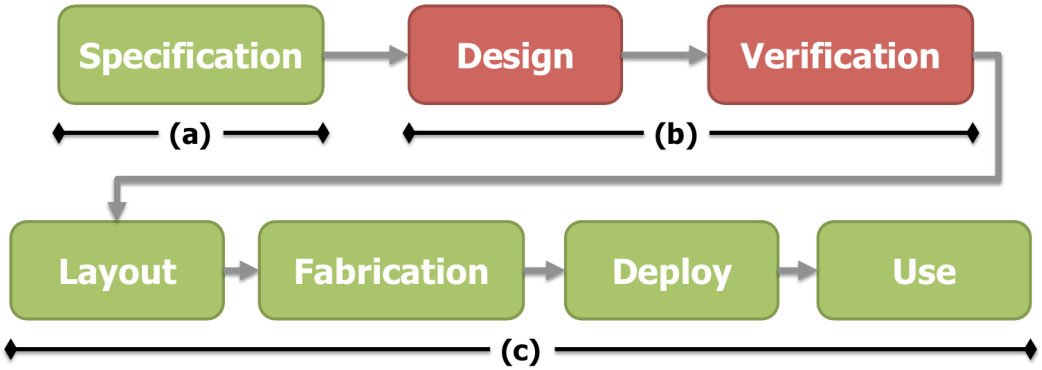


Figure 2.1: *Processor life-cycle. The green/light boxes represent stages that we trust, while the red/dark boxes represent stages that we assume are imperfect. We divide the flow into three parts: (a) The specification part, where trust is essential, because we base both recovery and the malicious circuit detection tool on the specification. (b) The functionality part, which we address here and (c) the supply chain part, which previous, orthogonal work addresses.*

2.2 Processor design flow

The life-cycle of a processor consists of several stages, starting from a set of requirements and an idea, ending-up in users' machines. Figure 2.1 shows each stage of the life-cycle, grouped into three broad categories. The green/-light stages represent trusted stages, while the red/dark stages represent untrusted stages; the focus of our work is validating trust in the red/dark stages. For the purposes of this dissertation, we group stages into three broad categories: specification, functionality, and supply chain. We describe the steps of each category, the different attack vectors, and how we validate trust for that category.

The first stage is in a processor's life-cycle is the specification stage: where the processor designer specifies the instruction set architecture (ISA), other functionality (*e.g.,* cache configuration), and the chip's physical constraints. Very rarely does this process start with a clean sheet; most specification stages start with an analysis of previous processors, increasing functionality and performance. This means the the specification stage is quick compared to the other stages, but often processor designers must revisit this stage to refine requirements based on the results of later stages. The result of this stage is generally a set precisely written documents in natural language.

The resulting specifications are the foundation for the rest of the stages and for our work. Since there is often little validation of the specification,

any errors—unintentional or otherwise—made in the specification stage are likely to carry over into the final physical processor and into our recovery and detection mechanisms. We, like the processor designers, blindly trust and abide by the specifications and thus add them to our TCB.

The second group of stages, denoted as *b* in Figure 2.1, is where processor designers create and verify the functionality mandate in the specification stage. Here, processor designers use a hardware description language (HDL) to create an executable version of the functional specification. The most popular HDLs are Verilog and VHDL. Both languages allow designers to express concurrent functionality at a medium level abstraction—registers (hardware state elements) and simple operations on data as it flows between registers—and directly deal with an abstract notion of time. After the initial coding phase completes, the cyclical process of functional verification and recoding starts. Functional verification often takes the form of simulating the design and running test cases on the processor. Testing of processor starts at the module level and completes with a whole system test that uses instructions as test cases. Some parts of the processor also undergo formal verification. Whatever the approach, it is intractable to completely test the entire processor.

The inability to completely verify a processor's functionality means that imperfections slip through to later stages. These imperfections may be bugs or malicious circuits: intent is the differentiating factor. Because this is the first practical target for inserting imperfections into the system and because imperfections from these stages make it into the other stages, we target our work here; we focus on protecting software from processor imperfections introduced during the design stage and missed during functional verification.

The final group of stages, denoted as *c* in Figure 2.1, covers everything required to get the functionality created in previous stages into the hands of the end user. The first stage in this group is layout. Layout starts with synthesis: converting a high-level description of the processor into an equivalent circuit consisting of only primitive elements (*e.g.,* transistors or gates). Now that the functionality is down to a level of abstraction amenable to future stages, processor designers need to layout the primitive circuit elements so they meet the area, power, and frequency constraints established in the specification stage. The processor is now ready for the fabrication stage. The fabrication stage is where the processor comes into physical form. This

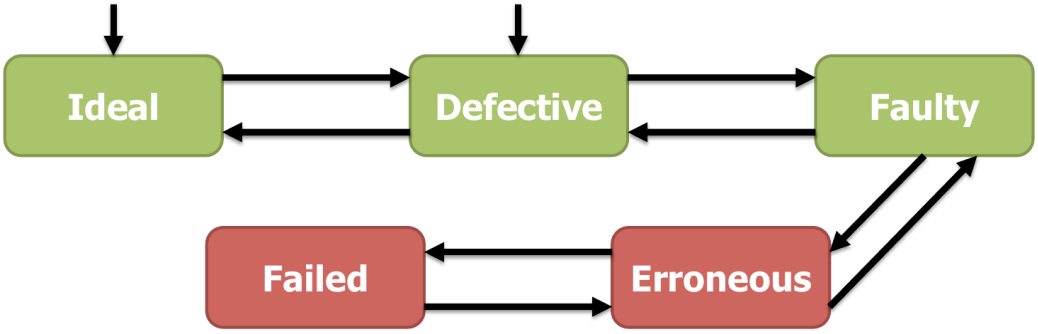


Figure 2.2: *System states and transitions that comprise a simplified version of the Multi-level Model of Reliability proposed by Parhami [32]. The red/dark states represent the levels where software is impacted.*

stage is very time consuming (second to verification) and the most expensive mostly due to the non-recurring cost of setting-up the tooling to produce a single chip. Before fabricated processors start their journey to end users, they go through a series of physical and functional tests to check for any flaws introduced during fabrication. The tests are not as complete as those done during the verification stage and there is less visibility into the the inner workings of the processor. After passing all the tests, the processor makes its way to the end user, where it is used.

Imperfections can come from bugs in the tools or through circuits added surreptitiously [23, 24, 25, 26], much like they would at design time, but at a lower level of abstraction. While there are many opportunities to insert imperfections into the processor in the final group of stages, we rely on previous work [12, 27, 28, 29, 30, 31] to increase trust.

2.3 Faults, errors, and failures

As shown in Figure 2.2, a processor fault is a defect in the processor that causes the processor to perform incorrectly at the gate level. Processor faults cause processor errors when the gate-level fault causes an inconsistent state at the ISA-level—meaning that the fault has impacted software. If the processor error prevents software from making forward progress, then we call this a processor failure.

There are two classes of processor faults: transient and permanent. Transient faults are processor imperfections that appear and disappear, with some

probability, over time. The most common model of transient fault used in research is the Single-Event Upset (SEU): where one bit of the hardware's state is flipped. The errors produced by transient faults are called soft errors, because a re-execution of the software sequence impacted by the error is unlikely to be impacted by the transient fault again.

Permanent faults, on the other hand, are likely to occur in the same way again and contaminate the same structures. Common sources of permanent faults are design errors, fabrication errors, and transistor fatigue. Permanent faults lead to hard errors (more so than transient faults lead to soft errors), which are more likely than soft errors to lead to processor failures [8]. Because of the dangers of permanent faults, they are the focus of our work.

2.4 Consistent state

We use the term *consistent state* as a shorthand for when the processor's ISA-level state, also known as software's state, is equivalent to the ISA-level state produced by a perfect implementation of the instruction set specification given the same instruction stream and starting ISA-level state. Without a consistent state, the same program could produce many different results, bewildering both users and software developers.

Ensuring that software always sees a consistent state is the primary goal of both reference implementations. While there are problems amenable to relaxing the predictability of software execution [33], we focus on creating a general-purpose mechanism that software can use in ad-hoc ways. We offer two systems, both powerful enough to reinforce the abstraction of a perfect processor (in spite of processor bugs and malicious circuits) and flexible enough to expose the opportunity for software to make its own reliability decisions.

2.5 Forward progress

For our purposes, forward progress is when software executes instructions after (program flow wise) the instruction that the detector associates with the imperfection activation. Executing instructions after the activating instruc-

tion is an indicator of forward progress because the detectors catch potential problems while the activating instruction (and its potentially contaminated state updates) is in-flight; therefore, any instruction that does manage to commit its state updates has been cleared by the detectors. Committing instructions after the activating instruction implies that the activating instruction has already been committed, meaning that the state of software has move forward.

Software is only be able to move forward if the recovery mechanism correctly moves the state of software past the imperfection. Thus, the sole goal of BlueChip is to create forward progress.

2.6 Recovery

Formally, recovery is when the processor avoids failure in the presence of processor errors. We define recovery as the ability of software to make forward progress, while maintaining a consistent state. In the two reference designs, we say that recovery is successful if given a starting state and some stream of instructions from software, the ending state of that the recovery mechanism produces and the ending state that a perfect processor would produce are equivalent.

Note that the ending states may not be identical. For example, it is still considered a successful recovery if the timer register differs. Since it is not a design goal to hide the side effects of recovery from software (although we do build mechanisms to protect the recovery firmware from modification by software), there is no need to update these types of implementation/environment-dependent state registers. In the case of the timer register, the ISA does not have a notion of time, so there is no expectation by software that an add takes N clock cycles.

Chapter 3

Related work

3.1 Recovering from processor errors

Recovery mechanisms allow the system to continue execution, in a consistent state, in the event of a fault detection. A general way of classifying a fault recovery mechanism is as either forward error recovery or backward error recovery. In forward error recovery, faults are corrected as part of the detection process. In backward error recovery, the state of the system must be restored to a consistent state before execution can resume.

Forward error recovery strategies, by their nature, require some form of duplication. Duplication can come in the form of additional hardware like DIVA, and its addition of a simple core, or in the form of additional software, as in SWIFT-R [34], with three versions of each program and a voter. While these and other forward error recovery techniques save time in the faulty case by not requiring rollback and re-execution, they require a significant slowdown in the bug free case and possibly large hardware area overheads. For the hardware bug fault model, the fault free cases is the common case.

For processor faults, current backward error recovery proposals rely on some level of checkpointing and rollback to ensure a consistent system state. Unlike forward error recovery, the overhead due to recovery can be significant due to state rollback and re-execution. Checkpointing also consumes CPU cycles in the fault free case by monitoring state changes and creating logs. SafetyNet [35] and CASPAR [36] are hardware implemented memory checkpointing and recovery mechanisms that support long-latency fault detection. Besides slowing the system down, these approaches add to the processor's complexity, increasing the time required for verification. Relax [37] on the other hand, implements checkpointing and recovery using a their custom Relax LLVM compiler. The Relax compiler removes any additional hardware

burden. The downside of Relax is that it only works for applications that can tolerate imperfect results and requires increased awareness and effort on the part of programmers.

Recovery proposals consist of techniques the rely on having system-wide checkpointing and rollback mechanisms [35, 36, 38, 39] and techniques based on intelligent recompilation of programs [40]. Checkpoint and rollback support adds a great deal of complexity to the system, delaying tape-out if hardware implemented, and causing significant run time overhead if software implemented. Checkpointing and recovery schemes also require awareness on the part of software to take advantage of the bug detectors. Intelligent recompilation schemes have a low run time overhead, but will not work for the large percentage of bugs not already patchable by software. Plus, intelligent recompilation also requires significant software support for dynamic recompilation and a complex central processor that is bug free.

Erratacator's fault recovery mechanism, while being classified as a backward error recovery mechanism, removes much of the overhead endemic to this class recovery by reusing the processor's own built-in checkpointing and recovery mechanism–the pipeline. Because Erratacator does *not* add any bubbles to the processor's pipeline, there is also none of the run time overhead associated with checkpointing systems.

Software implemented fault tolerance (SWIFT) [41] combines two identical versions of the same program together in the same instruction stream. This technique not only increases the fault free run time of programs, but due to large detection latencies, requires a rollback mechanism for recovery. Another software-only fault detection technique is MASK [34]. In MASK, the compiler generates invariants about the data values of a program, adding run time checks to ensure those invariants are upheld. Current implementations of MASK are limited to looking for known zero values, which, while low overhead, severely limits its ability to detect faults. A last software-only detection technique, triple redundancy using multiplication protection (TRUMP) [34], effectively generates a second version of each program by multiplying each data value by a constant, called AN-encoding. TRUMP has reduced run time overhead compared to SWIFT and improved fault detection over MASK, but without hardware support for increased register bit widths, AN-coding fails with large data values due to overflow. Also TRUMP, by its data-driven nature, is limited to a subset of an ISA's instructions.

3.2 Processor bug detectors

One hybrid approach for detecting processor faults proposes comparing the possible behaviors of a program with the actual behavior. Argus [38] detects faults by looking for divergence in the control flow, data flow, execution, and memory of a program compared to the all possible static flows and expected results. The techniques Argus uses for verifying execution are similar based on those discussed by Sellers *et al.* [42]. Much like DIVA, these compared execution-based techniques are complex, requiring additional hardware area, and increase verification effort, both of which Erratacator does *not*.

Another hybrid processor fault detection approach is to periodically pause the processor and check for an inconsistent state. This is the central technique employed by SWAT [8], ACE [43], and the proposal of Shyam *et al.* [44]. These techniques impose a large run time overhead in the bug free case and, due to the latency of software-level detection, require a checkpointing and rollback mechanism for recovery. These techniques also require complex hardware to support accurate detection of faults not visible to software. Even with hardware support, these techniques are likely to miss bug activations that happen below the micro-architectural level, since there hardware detectors are static and only look at signals at that level.

3.3 Hardware attacks

The available examples of attacks on hardware are currently limited to what academia produces as those in industry speak only aloofly about attacks they have seen, never coming close to describing the attack's behavior, or even the attacker's motives. Even in academia there exists a limited pool of example attacks, mostly coming from work by King *et al.* [4] and Hicks *et al.* [45].

Hadvzic *et al.* were the first to look at what hardware attacks would look like and what they could do [46]. They specifically targeted FPGAs, adding malicious logic to the FPGA's configuration file that would short-circuit wires, driving logic high values, in an attempt to increase the device's current draw, causing the destruction of the device through overheating. They also proposed both a change to the FPGA architecture and a configuration analysis tool that would defend against the proposed attacks.

As a part of their paper describing their malicious circuit fingerprinting approach, Agrawal *et al.* describe three attacks to RSA hardware [12]. One attack uses a built-in counter which after a pre-described number of clock cycles, shuts down the RSA hardware. The other two attacks use a comparison based trigger which, when activated, contaminates the results of the RSA hardware. The attacks show how small of a footprint targeted attacks on hardware can have in terms of circuit area, power, and the amount of coding effort required to weaken hardware.

The Illinois Malicious Processor (IMP) by King *et al.* [4] is the first work to propose the idea of malicious hardware being used as a support mechanism by attack software, usurping software-level security primitives. These intentional hardware security vulnerabilities, inserted during design time, are termed footholds. Being small in terms of number of lines of code and effect on the rest of the design, footholds are shown to be difficult to detect using conventional means or side channel analysis. IMP contains two attacks, unauthorized memory access, where user processes can access restricted memory addresses and shadow mode, where the the processor executes in a special, hidden, mode. Three malicious software services that leverage the inserted footholds, privilege escalation, a backdoor into the login process, and a password stealer are constructed. In follow-on work, Hicks *et al.* [45] reimplemented the attacks and verified that the attacked hardware passed SPARCv8 certification tests.

Jin *et al.* developed eight attacks of the staged military encryption system codenamed Alpha [47]. The attacks corrupted four different units and three of the data paths of the encryption system, exposing the vulnerability of current hardware to malicious insertions. The eight attacks ranged in area overhead from less than 1% to almost 7% while still managing to pass functional verification tests. The results re-enforced the notion of small, buried, but powerfully attacks from King *et al.*'s previous work.

3.4 Defenses to hardware attacks

This section covers research on detecting and defending against malicious hardware during design time, possibly with a run time component.

3.4.1 Design time

In Moats and Drawbridges [16], Huffmire *et al.* propose an FPGA isolation primitive along with a cooperating inter-module communication philosophy in an attempt to bring the properties that a memory management unit brings to software processes to distinct hardware units. The isolation primitive requires a perimeter of unused logic elements, moats, around each independent design unit. The inter-module communication philosophy, drawbridges, requires the use of predefined, statically verifiable, pathways for communicating between distinct hardware units. While Moats and Drawbridges does allow distinct hardware units to have data integrity and confidentiality, moats can dramatically increase area of a design with many units that use most of a FPGA's logic resources. More importantly, moats and drawbridges do *not* increase trust in any individual unit or in the design as a whole.

In research targeted at the post manufacturing stage, but directly applicable to the design stage, Agrawal *et al.* [12] propose a signal processing-based technique for detect additional circuits through side-channel power analysis. This approach faces two key challenges. First, it assumes that the defender has a reference copy of the design without a trojan circuit, an assumption that breaks if there is no pre-existing unit. Second, the experimental results are from simulations on small (1000 gate) circuits. Thus, it is unclear how well the proposed technique will work in practice, on large circuits, such as microprocessors.

Formal methods such as symbolic execution [48, 49], model checking [50, 51, 52], and information flow [53, 54] have been applied to software systems for better test coverage and improved security analysis. These diverse approaches can be viewed as alternatives to UCI, and may provide promising extensions for UCI to detect malicious hardware if correct abstractions can be developed.

3.4.2 Run time

TrustNet and DataWatch [55] work in-conjunction at run time, ensuring that an untrusted hardware unit does not generate more or less output data than it is supposed to, given the inputs it sees. This approach, based on the DIVA architecture [10], consists of hardware monitors architected as a series

of triangles where the inputs and outputs of each untrusted hardware unit are monitored by the other two units of the monitor triangle. The units of a given monitor triangle are consecutive, hence cooperating units in the processor's natural pipeline. Given an input, the monitor determines what the appropriate amount of output is, signaling an error if too little (denial-of-service) or too much (information leakage) data is produced. This approach works well in an environment where a processor is outsourced and the system integrator has enough knowledge and ability to create and insert the monitor triangles. It is also unclear how this approach handles mutated data.

Fault-tolerance techniques [56, 57] may be effective against malicious ICs. In the Byzantine Generals problem [56], Lamport, *et al.* prove that 3m+1 ICs are be needed to cope with m malicious ICs. Although this technique can be applied in theory, in practice this amount of redundancy may be too expensive because of cost, power consumption, and board real estate. Furthermore, only 59 foundries worldwide can process state-of-the-art 300mm wafers [58], so one must choose manufacturing locations carefully to achieve the diversity needed to cope with malicious ICs.

In some respects, the BlueChip system resembles previous work on hardware extensibility, where designers use various forms of software to extend and change the way deployed hardware works. Some examples of this type of hardware extensibility include patchable microcode [59], firmware-based TLB control in Itanium processors [60], Transmeta code morphing software [61], and Alpha PAL code [62]. Our design uses some of the same hardware mechanisms used in these systems, but for coping with malicious hardware rather than design bugs.

3.4.3 Supply chain

Analog side effects can be used to identify individual chips. Process variations cause each chip to behave slightly different, and Gassend, *et al.* use this fact to create physically random functions (PUFs) that can be used to identify individual chips uniquely [11]. Chip identification ensures that chips are not swapped in transit, but chip identification can be avoided by inserting the malicious circuits higher up in the supply pipeline.

The possibility of using power analysis to identify malicious circuits was

considered in [12]. However, power analysis began as an attack technique [63]. Hence there is a large body of work to preventing power analysis, especially using dual-rail and constant power draw circuits [13, 14]. For the designer of a Trojan circuit, such countermeasures are especially feasible; the area overheads only apply to the small Trojan circuit, and not to the entire processor.

One can obtain the layout of an IC and determine if it matches the specification. IC reverse engineering can re-create complete circuit diagrams of manufactured ICs [15]; the defender can inspect their circuits to verify them. Unfortunately, such reverse engineering is time consuming, destructive, and expensive. It may take up to a week and $250,000 to reverse engineer a single chip [15].

3.5 Formal analysis of hardware

Hardware, due to its limited resources and cycle-based behavior is generally more amenable to formal analysis than software. The current focus of the majority of research on formal methods, as applied to hardware, centers on verifying that the hardware faithfully implements some specification (verification).

Model checking is the process of verifying that the behavior of a hardware design matches properties specified using temporal logic formulas [64, 65]. The drawback of model checking is computational complexity through state space explosion [66]. Even with advances in Boolean satisfiability solving (SAT) [67] and symbolic representation [68], the most advanced tools are unable to completely tame the state space explosion problem enough to handle large sequential designs [69, 70, 71, 72, 73, 74, 75, 76]. Also, writing a complete set of properties that precisely verify every aspect of the hardware design is of the same complexity as describing it in using hardware description languages, meaning the same likelihood of errors and increased time and money investment.

Chapter 4

Erratacator

Processors are *not* perfect—they have bugs. Modern processors are as complex as modern operating systems, consisting of millions of lines of code [1], yielding chips with billions of transistors [2]. Since completely verifying processors of such complexity is intractable, verification tools leave hundreds of bugs unexposed in production hardware. For example, the errata document for Intel's Core 2 Duo processor family [3] contains information on 129 known bugs. Averaged over the production life of the processor family, that is more than 2.5 bugs per month. Yet despite these imperfections, we design, debug, and deploy software systems trusting that processors are bug free.

Some processor bugs force designers to change the way they implement features. For example, GDB designers use breakpoints rather than instruction overwriting to interpose on the execution of a thread for their "fast tracepoint" implementation. This lower performance design comes from the undefined system behavior resulting from processor bugs [5].

Processor bugs can also cause security vulnerabilities. For example, the MIPS R4000 processor has a bug that enables user-mode code to control the location of the exception vector, giving attackers the ability to run user-mode software at the processor's supervisor level [6]. In addition, the designers of Google's NativeClient [7], which is browser-based sandbox for native x86 code, argue that their reduced attack surface is a result of limiting the instructions that NativeClient modules can execute, thus avoiding processor bugs. The designers of NativeClient contrast their design to Xax [77], which the NativeClient authors claim has security vulnerabilities caused by processor bugs.

One approach to patching processor bugs is to add redundant execution in the hopes that at least one of the executions will be bug free. Diva [10] is a redundant execution-based approach to detecting and recovering from processor bugs (among other processor faults) that adds redundancy to the

hardware layer in the form of a second processor from a previous generation. Presumably, the previous generation processor is less buggy, which allows Diva to mark any execution that diverges from the previous generation processor's execution as bug contaminated. Presumably, the previous generation processor is also slower, as the previous generation processor must sign-off on the execution results of the current generation processor before Diva commits the results. This limits software to the execution speed of the older, slower processor; imposing run time penalties even when no bug activations occur. Another disadvantage of Diva is the open question of handling instructions and other features present in the current generation processor not present in the previous generation processor. But, the most impactful drawback of Diva is the doubling of hardware area and added complexity of connecting the two processors together, which serves to increase the risk of processor bugs.

We propose combining the low latency of hardware-based processor bug detection with the dynamic power of software-based recovery to form a practical processor bug detection and recovery platform we call Erratacator. Separating Erratacator from previous proposals is the insight that processors have sufficient innate functionality both to maintain a consistent state in the face of bug activations and to allow software to make forward progress in the presence of processor bugs. Erratacator maintains a consistent state by connecting low latency processor bug detectors [9] to the processor's pipeline flush controller. This allows the processor to simply flush any contaminated state, obviating any need for the heavyweight checkpointing and rollback mechanisms required by software-based bug patching approaches. To eliminate the complex and hardware intensive recovery requirements of hardware-based processor bug patching approaches, Erratacator employs flexible, formally verified, software-level, recovery routines that recode instruction streams in an effort to route execution around processor bugs. The insight driving this technique is that processors contain large amounts of redundant functionality and that the vast majority of a processor's functionality is bug free.

By repurposing processor functionality, Erratacator reinforces the abstraction of a perfect processor without placing any burden upon software developers and with trivial increases in hardware area and complexity for processor vendors. If software developers want to explore the trade-off between performance and reliability, Erratacator exposes an instruction set architecture

(ISA) extension that software to manage which bug detectors are active. In the same vein, Erratacator enables processor vendors to explore the trade-off between design time verification and run time overhead, possibly parallelizing tapeout and the final, time inefficient [78], stages of debugging.

Experiments with Erratacator, using publicly documented bugs [79, 80, 81], show that Erratacator can detect and recover from a wide range of bugs. The hardware area overhead imposed by Erratacator's bug detectors is less than 1% and the software run time overhead is less than 12% as long as bug detections occur separated by 8000, or more, instructions. Interesting results include that Erratacator can recover from most, but not every processor bug and that providing practical protection in the presence of many processor bugs requires software intervention.

In summary, the contributions of this reference implementation are:

- We design, implement, and evaluate a complete and practical system that detects and recovers from real processor bugs.
- We use the processor pipeline's flush mechanism to eliminate the heavy-weight checkpointing and rollback mechanisms required by previous proposals.
- Instead of eliminating functionality or requiring additional processors, we use the dynamic power of software and the redundant, bug free functionality inherent to processors as the recovery mechanism.
- We use formal verification to prove that the architecture models employed by Erratacator firmware (*i.e.,* some instructions removed from the ISA) and the model described by the processor's ISA are equivalent.
- We are the first to implement a recent processor bug detection proposal, expose scaling issues, and propose and implement mechanisms that allow better scaling.

4.1 Erratacator overview

Processor bug activations are rare, but pervasively powerful. An ideal patching solution transparently reinforces the abstraction of a perfect processor

while imposing no run time overhead in the common case of no bug activations. Any approach to patching processor bugs must have (1) the ability to detect bugs and (2) a recovery mechanism that allows for safe and correct software execution post-bug activation. Additionally, the detection and recovery mechanisms must work together to ensure that the transition from detection to recovery removes all state contaminated due to the bug activation.

In light of these goals, this section presents the design principles and assumptions behind Erratacator, a hybrid platform for detecting and recovering from processor bugs.

4.1.1 Design principles

Three key principles guide the design:

1. *Minimize recovery and bug detection complexity.* For Erratacator to be practical and not burden hardware designers, we must avoid adding complex hardware structures by repurposing existing mechanisms.

2. *Implement correct recovery firmware.* For Erratacator to be practical and not burden software designers, we must have high assurance that the recovery firmware pushes software state forward correctly, even when running on buggy hardware.

3. *Maintain current ISA abstractions.* For software running on Erratacator, including hypervisors and operating systems, Erratacator must provide the illusion of executing on a perfect processor.

4.1.2 The Erratacator approach

Figure 4.1 provides an overview of the components that comprise Erratacator and their responsibilities. Erratacator consists of hardware-implemented processor bug detectors and firmware-level processor bug avoidance routines. Note the detectors only interact with the processor through a single wire to the exception logic, not adding to the complexity of the processor. Also note that the software layer remains unaltered, unaware of any processor imperfections or protections.

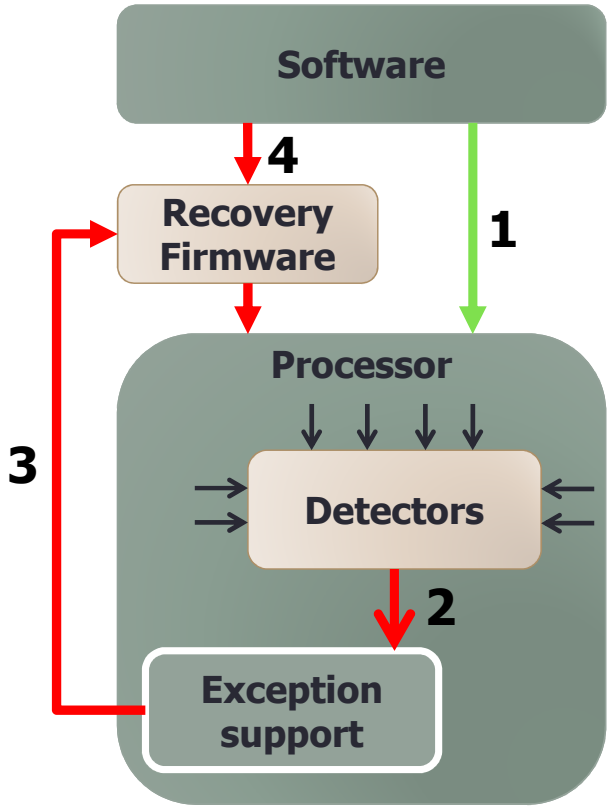


Figure 4.1: *Flow of detection and recovery in Erratacator: 1) Software executes normally on the processor while the bug detectors check the processor's state for bug signature matches 2) A bug signature match causes the bug detectors to signal the processor for an exception 3) The exception causes any bug contaminated state to be flushed and control passes to the recovery firmware 4) The recovery firmware backs-up the ISA-level state of the processor, then fetches, decodes, recodes, and executes instruction streams from the software layer 1) Eventually the recovery firmware loads the updated ISA-level state of the software layer back into the processor from the simulator and passes control back to the software layer.*

The hardware portion of Erratacator's recovery repurposes the processor's built-in exception handling logic as a checkpointing and rollback mechanism, avoiding the complex and heavy-weight hardware structures required by other proposals [35]. A key observation is that processor exception handling mechanism already provides a version of checkpointing and rollback, just with a checkpoint window spanning from the fetch stage to right before the instruction's changes are committed to the ISA-level state (*e.g.,* three stages on the traditional MIPS 5-stage processor). When a bug detection exception occurs, the processor flushes in-flight state and passes control to

a firmware-implemented exception handler, ensuring that the architecturally visible processor state remains consistent. Because the checkpoint window is only a few cycles, it requires processor bug detection latencies equally small to ensure a consistent software state.

To meet such latency demands, Erratacator's bug detectors reside in hardware, constantly monitoring the processor's hardware-level state for bug activations, as show in Figure 4.1. Erratacator's detectors do this by comparing the current processor state values (*i.e.,* the value of all flip-flops that implement the processor) to those in a dynamically definable signature. If no bug signatures match the current state of the processor, execution continues. When a signature does match, the detection hardware triggers an exception, which the processor handles in a similar fashion to any other exception.

The processor delivers the bug detection exception to Erratacator's recovery firmware. The recovery firmware first backs-up the processor's state, allowing for processing of recursive bug detection exceptions. Then the recovery firmware loads and recodes a series of software layer instructions, in a attempt to advance correctly software state by avoiding re-triggering the bug. When done, Erratacator's recovery firmware updates the state of the processor with the software's simulated state and passes control back to the software layer.

One key aspect of the recovery firmware is that it simulates instructions using a different sequence of instructions to avoid re-triggering the bug—think of it as trying to achieve the same ISA-level effect as the original instruction stream, but restricting the set of available instructions. For example, Erratacator simulates multiply instructions using shifts and adds. To ensure correctness of the recoding, we use formal methods to prove equivalence between the net effect on ISA-level state of the original instruction stream and the recoded instruction stream. Formal verification does not ensure ensure that the ISA used for recovery is as expressive as the original ISA, so recovery is not always possible. In such cases, the flexibility of firmware allows for patches to the recovery routines.

4.1.3 Assumptions

For Erratacator to work, we make three assumptions. First, the processor is in a state that can be interrupted by a bug detection exception. This assumption holds for all of the supervisor-mode and user-mode software that runs on the selected processor, even with interrupts disabled. As demonstrated in Section 4.5, processor bugs can violate this assumption even if software does not, preventing complete recovery.

The second assumption is that the processor's pipeline flushes all bug-contaminated state upon receiving an Erratacator exception. Because Erratacator repurposes the processor's pipeline as a checkpointing and rollback mechanism, it relies on proper operation of the flush mechanism to maintain a consistent state in the face of processor bug activations. Processor bugs that violate this assumption, as shown in Section 4.5, can prolong recovery or even make it impossible.

The third assumption is that the recovery firmware's entry and exit routines execute without triggering a processor bug. To allow for handling of recursive bug detections, the first and last action the recovery firmware performs is backing-up to and restoring from memory the state overwritten by the processor when it handles an exception (*e.g.,* the Status Register). If the recovery firmware's entry or exit routines trigger a processor bug, it is possible the processor will live lock, perpetually attempting to recode the same instruction sequence. However, the entry and exit routines are 14 instructions each and guaranteeing that they run to completion means that recursive Erratacator exceptions are tolerable at any time.

4.2 Recovery firmware

Erratacator firmware is responsible for helping software safely execute sections of code that trigger processor bugs. Erratacator's recovery firmware pushes software state forward through the use of bug avoidance routines which recode the software's instruction streams using a different set of instructions, *i.e.,* a sub-ISA. By changing the set of available instructions, the firmware mutates the original instruction stream to one that has the same ending ISA-level state, but which takes a different path to get there. This protects against reactivating both instruction and state sensitive bugs.

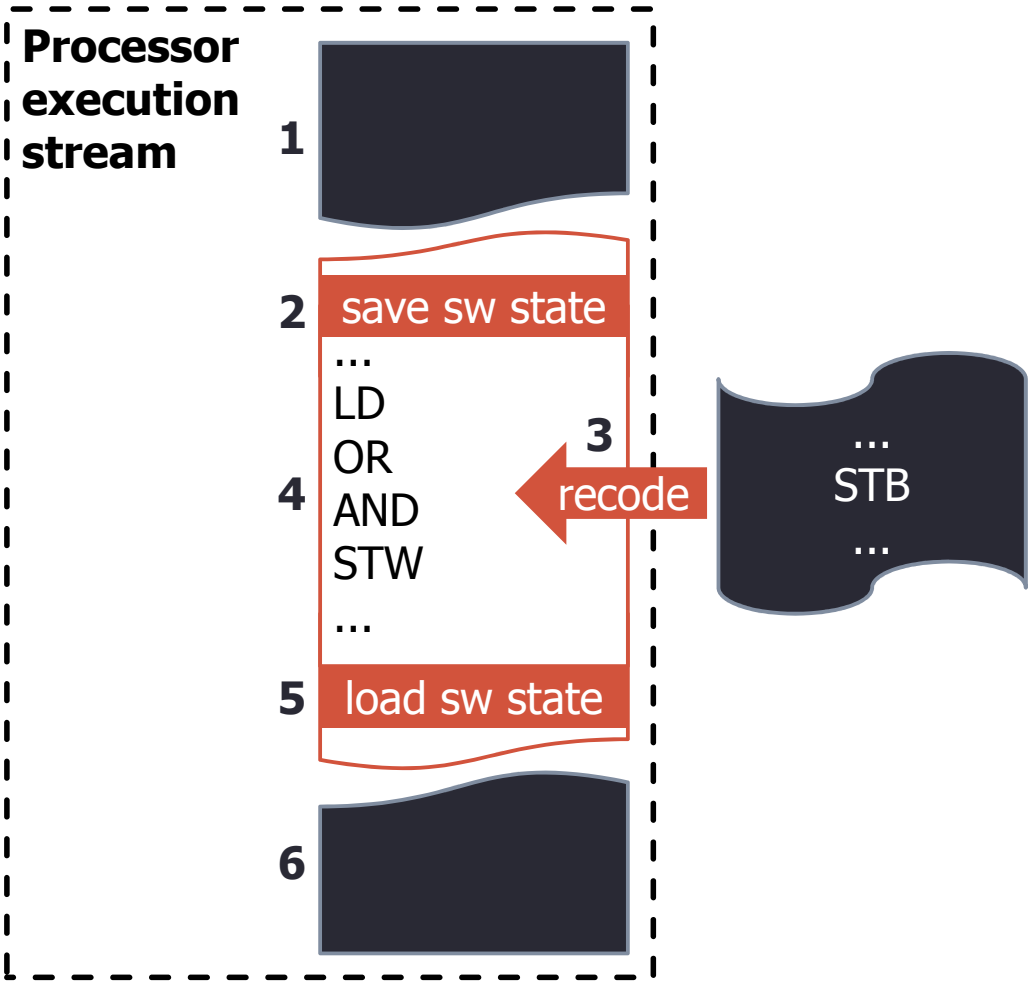


Figure 4.2: *1) Software executing normally on the processor 2) When a bug detection occurs, control passes to the recovery firmware which backs-up the software's state and exception registers in pre-defined simulator data structures 3) The recovery firmware fetch, decodes, recodes, and 4) executes several instructions from the software level 5) Once past the bug, the recovery firmware loads the processor with the software state in the simulator and passes control back to the software level 6) Software resumes normal execution*

Figure 4.2 provides an overview of how the recovery process works to push software execution forward, around processor bugs by recoding the original instruction stream and executing the recode stream on the processor. Erratacator's recovery firmware consists of entry and exit routines, routines that fetch and decode instructions from software, and recoding routines.

4.2.1 Hardware/firmware interface

One of Erratacator's goals is to not increase the complexity of the processor, some added complexity is required to support software recovery. It is important for efficient recovery that the firmware know the source of the bug detection. This requires the addition of a ISA-level register that the firmware can read to direct recoding. Another issue is that bug detections can occur even in software that is executing with interrupts disabled. This means that invoking the firmware could overwrite special purpose registers not backed-up by software. To solve this issue, Erratacator adds ghost versions of all exception registers and creates an additional ghost register to be used as a temporary holding place for the recovery firmware's entry routine—we usually put software's stack pointer there. Without this extended interface between hardware and the firmware it would be impossible to invoke the recovery firmware without losing some crucial data.

Another addition to the hardware/firmware interface is a set of registers that allow for fine-grained control of the general purpose bug detection fabric. These registers include the system mask, system signature, and a bug enable register that controls which bug detectors can fire exceptions. This is essential for allowing Erratacator to keep high layers of the system protected from newly discovered bugs and allows for a more nuanced control of where Erratacator is executing in the protection vs performance tradeoff space. These bug detector control registers also make it possible for software to control which bug detectors are active and when (*e.g.,* operating system vitalized signatures and masks) as discussed later in this section.

4.2.2 Firmware/software interface

When the processor issues an bug detection exception, Erratacator's recovery firmware is tasked with pushing the state of the software layer forward, so that it can continue executing on its own, past the bug activating code. After the recovery firmware saves the current state of the software layer (as stored in the processor), initializes itself, and determines what the bug is, it is ready to assist the software layer. This requires that the firmware reach into the address space of the software layer to read instructions and data values. The recovery firmware does this by manually walking the TLBs (if the MMU was

enabled at the time of detection) and fetching the required value straight from memory.

Erratacator firmware then simulates the interrupted software using a different set of instructions to avoid re-exercising the buggy processor logic. Although our simulation is general purpose and our technique of simulating instructions using a different set of instructions enables Erratacator to avoid a wide range of bugs, our hardware/firmware interface, by exposing which bug caused the Erratacator exception allows for custom recovery routines. The custom recovery routines are more efficient that our general purpose simulator, but the simulator is at least a reliable backstop.

4.2.3 Handling recursive bug detections

Recovery in Erratacator is a trial and error process. Because an attempt at recovery may activate a bug itself, Erratacator is designed to handle recursive bug detections—outside of the entry and exit sequences. This allows the recovery firmware to keep changing the sub-ISA used to recode the original instruction stream, increasing the odds of recovery. The ability to handle recursive bug detections also allows recovery firmware designers to sacrifice likelihood of recovery in favor of lower average run time overheads; since they can also try the safer recovery method if the faster method fails. Experimental results in Section 4.5 highlight the difference in speed of two different recovery approaches.

The first obstacle to handling recursive bug detections is protecting the ISA-level state that taking and exception atomically updates. Erratacator handles this by creating a special stack for all atomically updated registers. The recovery firmware stores a pointer to the most recent frame at a known address in it memory space. Since the frames are all the same size, traversing between the frames of different recovery firmware invocations it trivial. All atomically updated registers and the pointer are saved/updated in the entry routine and the most recent frame is unloaded into their associated registers in the processor and the pointer decremented in the exit routine.

The second obstacle is that the recovery firmware cannot simulate itself. This creates problems when bug detections occur while the recovery firmware is fetching or decoding instructions from software, *i.e.,* not actually recod-

ing or executing the recoded instruction stream. In these cases, which the recovery firmware knows by inspecting the address associated with the bug detection, recovery will just restart in hopes that the act of taking the exception was enough of a disturbance to avoid the bug. To avoid live locks due to this restriction, we avoided complex control paths in the fetch and decode code to make its execution pattern very regular. An inspection of the documentation of the errata-like bugs from our processor revealed that none threatened the fetch or decode parts of the recovery firmware.

If the recovery firmware is recoding or executing the recoded instruction stream, it will restart the simulation process, but with a different recoding algorithm.

4.2.4 Software managed reliability

Results from experiments (Section 4.5), where Erratacator was tasked with monitoring an increasing number of bugs, expose the problem of false bug detections. Bug detector resources are limited (Section 4.3), therefore, when Erratacator needs to monitor for activations of multiple bugs, there is a chance that the bug signatures will involved the same state values. To prevent contamination due to missed bug activations, Erratacator must combine competing bug signatures in a pessimistic way, creating the potential for false positives.

Dynamically enabling and disabling bug detectors can reduce false detections while maintaining a consistent ISA-level state. Since bug signatures in Erratacator are dynamically reconfigurable, both hardware and software have an opportunity to manage which bugs detectors are active. In the case of software, Erratacator extends the processor's interface with software, allowing software to manage its own reliability and run time overheads by controlling which processor bugs it's exposed to and when. Experimental results validate the power of this mechanism to reduce false detections, increasing software performance.

4.2.5 Proving correctness

This section covers our formal verification effort on the recovery firmware. Our goal is to prove that the recoding routines used to route software execution around processor bugs are equivalent to the instructions that they replace. Note that we assume that the firmware's entry and exit routines are correct and that the firmware correctly fetches instructions from the software stack. We also assume the firmware starts in a consistent ISA-level state.

Because Erratacator firmware simulates instructions *without* using the instruction it is simulating, the recovery firmware is more complex than a traditional instruction-by-instruction simulator. For example, we simulate *ADD* instructions using a series of bit-wise Boolean operations, shifts, and comparisons to check for carry and overflow conditions. These bit-manipulation operations are difficult to reason about manually, which motivates our use of formal methods to prove the correctness of our implementation.

The code that we prove falls into three categories. First, we prove correct helper functions that we compose together to simulate instructions. These functions include sign extend, zero extend, and overflow and carry logic for arithmetic operations. Second, we prove correct alternative implementations of instructions that trigger processor bugs. These instructions include *DIVIDE*, *MULTIPLY*, *ADD*, and *SUBTRACT*. Here we prove that using a set of completely different instructions is equivalent, at the ISA-level, to the bug inducing instruction. Third, we specify invariants on our processor state structures to specify and prove correct our instruction decode logic. All of our source code, with our specifications, can be found on our web site [82].

To verify our implementation, we used VCC [83]. VCC enables sound verification of functional properties of low-level C code. It provides ways to define annotations of C code describing function contract specification. Given the contracts, VCC performs a static analysis, in which each function is verified in isolation, where function calls are interpreted by in-lining their contract specifications. To create the contracts for each instruction's function within the simulator, we used the description from the OR1200's instruction set specification. The instruction set specification provides precise ISA state changes given an instruction and the current ISA-level state. Our code contracts look very similar, with assertions on function inputs acting as checks on the starting ISA-level state and assertions that check the final ISA-level

```
static void calc_divu(u32 divisor, u32 dividend, u32 *quo_out, u32 *rem_out)
{
    u32 idx, quotient, reminder;
    // i32 rem; // BUG HERE
    i64 rem; // GOOD
    u64 rq = dividend;

    rq <<= 1;

    for(idx = 0; idx < 32; idx++)
    {
        rem = (i64) ((rq >> 32) & 0xffffffff);
        rem = rem - divisor; // BUG expressed here
        if(rem < 0) {
            rq <<= 1;
        } else {
            rq = (rq & 0xffffffff) | (((u64) rem) << 32);
            rq <<= 1;
            rq |= 0x1;
        }
    }
    ...
}
```

Listing 4.1: *Divide bug 1*

state for the expected update. Therefore, we consider verification complete if for every instruction in the ISA, the input and output state of the simulator matches the states mandated by the instruction set specification.

Our verification efforts, which took place after all conventional software tests were passed, revealed subtle bugs in our sign extend helper function (discussed later) and our divide instruction implementation that were caused by implicit, yet incorrect, assumptions that we made when writing the simulator (mostly to do with signed vs unsigned numbers).

Formal verification exposed two bugs in our divide recoding function, calc_divu. The first one, shown in Listing 4.1, is caused by representing the difference between the divisor and the most significant bit of the dividend using a signed 32 bit integer (the variable named "rem" in the implementation). If the dividend is larger than 2^{31}, the calculated difference stored in "rem" is positive, while the correct difference is negative. So, the function wrongly increments the quotient: instead of just shifting left. For example,

```
static void calc_divu(u32 divisor, u32 dividend, u32 *quo_out, u32 *rem_out)
{
    u32 idx, quotient, reminder;
    // u32 rem; // BUG HERE
    i64 rem; // GOOD
    u64 rq = dividend;

    rq <<= 1; // BUG partially activated

    for(idx = 0; idx < 32; idx++)
    {
        rem = (i64) ((rq >> 32) & 0xffffffff); // BUG fully activated
        rem = rem - divisor;
        ...
    }
    ...
}
```

Listing 4.2: *Divide bug 2*

the procedure will not correctly compute the quotient and the reminder of $1/((2^{31})+1)$.

The second bug—which we created while trying to fix the first bug—shown in Listing 4.2, is due to trying to fit the reminder in 31 bits, because we used a signed integer. Because of the initial left shift and 32 left shifts in the loop, the reminder is stored in upper 31 bits of the register pair reminder-quotient, denoted "rq" in the listing. That the reminder is in the upper 31 bits is also confirmed when shifting right by 33 when obtaining the final value of the reminder. This bug comes up when the divisor is greater than the dividend, and the dividend is greater than $(2^{31})-1$. For example, the procedure will return the reminder 0 instead of 2^{31} when computing $2^{31}/((2^{31})+1)$.

Interestingly, these division bugs are reminiscent of a processor bug that has eluded several attempts at patching by the OR1200 community [80] (see Section 4.4). They both are related to the same signed vs unsigned number representation issues.

Aside from verifying the functional equivalence of our recoding routines, we verify parts of the decode logic responsible for splitting instructions into meaningful parts, *e.g.,* opcode, ALU operation, and immediate. The difficulty is how to split an instruction depends on what the instruction is and even parts with the same label, *e.g.,* immediate, are different sizes for differ-

```
struct control {
  u32 opcode; _(invariant \this->opcode
                          == (\this->inst >> 26) & 0x3f))
  u32 alu_op; _(invariant \this->alu_op < (1<<10))
      ... };
```

Listing 4.3: *Instruction splitting assertions*

```
static void maci(struct CPU *cpu, struct control *cont)
{
    m = cpu_get_mac(cpu);
    a = cpu_get_gpr(cpu, cont->rA);

    // imm = sign_extend(cont->I, 10); // BUG
    imm = sign_extend(cont->I, 15); // GOOD

    calc_mul(a, imm, &r);
    m += tmp;
    cpu_set_mac(cpu, m);
}
```

Listing 4.4: *Incorrect immediate size due to copy-and-paste bug*

ent instructions.

To prevent errors in splitting instructions we first consult the instruction set specification to determine the names and sizes of all possible pieces of an instruction. We then add this information, in the form of invariants, to the matching structures in our simulator. Listing 4.3 shows an example of how we assert the size of instruction pieces in our simulator.

There is still a problem of different sized instruction pieces with the same label. For example, the immediate piece of an instruction can be 11-bits or 16-bits wide. We need a way to prove that instructions expecting a 16-bit immediate get one, while instructions expecting an 11-bit immediate get one. Listing 4.4 shows a copy-and-paste error where we used a 16-bit immediate as an 11-bit immediate. To catch these types of bugs, we add preconditions to every simulator function that check for incorrectly sized inputs from the decode stage. Listing 4.5 shows the preconditions added to the sign extend function that caught the mis-sized immediate.

```
static i32 sign_extend(u32 value, u32 sign_bit)
    _(requires sign_bit < 31)
    _(requires value < (1<<(sign_bit+1)))
    _(ensures (to_u32(\result) & maxNbit32(sign_bit+1)) ==
                        (value & maxNbit32(sign_bit+1)) )
    _(ensures !bit_set(value, sign_bit) ==> \result >= 0)
    _(ensures !bit_set(value, sign_bit) ==> (to_u32(\result) == value))
    _(ensures !bit_set(value, sign_bit) ==> ( \result == to_i32(value)))
    _(ensures bit_set(value, sign_bit) ==> \result < 0)
    _(ensures bit_set(value, sign_bit) ==>
              ((to_u32(\result) & maxNbit32(sign_bit+1)) == value))
    ...
```

Listing 4.5: *sign_extend preconditions*

4.2.6 Approach weaknesses

Complete recovery is not always possible, even if all the assumptions in Section 4.1 hold. Generally, if there is insufficient bug-free redundancy available or the bug avoidance routines do a poor job at exploiting bug-free redundancy, recovery will fail. One concrete example for the OR1200 (see Section 4.4) is a bug involving a write or read from the byte-bus. As its name implies, the byte-bus is an 8-bit wide peripheral bus that can only be accessed at the byte level. If there is a flaw in the processor involving byte-bus accesses, there is no way for Erratacator firmware to recode execution around the bug.

Possible approaches to handling this type of failure includes simply updating the recovery firmware or even extending the ISA further to allow Erratacator to communicate cases of incomplete recovery to the software level. Another option is for processor designers to ensure that functionality that has the least redundancy is verified the most.

4.3 Detecting processor bug activations

Bug detectors are responsible for alerting the recovery mechanism to processor bug activations. The detection latency of a bug detector is the time difference between when the bug produces a fault and when the detector detects the error caused by the fault. The larger the detection latency, the

more system state is corrupted by the activation of the bug.

Low-latency bug detection is the cornerstone of Erratacator. By detecting a bug before the write-back stage (where instruction results commit to ISA state), Erratacator can forgo the traditional heavyweight checkpointing and rollback mechanisms and instead use the processor's pipeline and flush operation to ensure the processor maintains a consistent state. This requires the detection of all processor bugs in less than four cycles.

In designing detectors that operate at such low detection latencies, the method used to describe bugs is key. The description method employed by designers determines not only what bugs are detectable, but the number of false activations, and the hardware area consumed by the detectors.

To balance these concerns, Erratacator adapts a previously proposed hardware bug detection mechanism, created by Constantinides *et al.* [9]. Their bug detection mechanism has shown that it can provide single cycle detection, with modest hardware requirements (10% area overhead), while being able to detect the majority of the bugs in a processor. Section 4.3.1 provides an overview of their bug detection algorithm. While we do not attempt to evaluate or improve upon their bug detection approach, we do make subtle changes worth documenting and provide the first implementation as a part of our Erratacator platform. Section 4.3.2 provides details on the changes required to the detection mechanism to allow the processor's pipeline to serve as the checkpointing mechanism.

4.3.1 Background

Figure 4.3 provides a high-level overview of the components and connections involved in the bug detection mechanism proposed by Constantinides *et al.*. The basic steps involved are signal selection, hardware creation, bug signature creation, and bug detection. In general, signatures are a string of bits that match processor states, masks are values that Constantinides *et al.* use to specify relevant value bits, and segments are units of four bits that they use to build up an overall signature.

During design time, the processor designer selects a set of signals in the circuit to monitor for bugs. Constantinides *et al.* advocate for selecting signals that are driven directly from flip-flops, which reduces hardware area

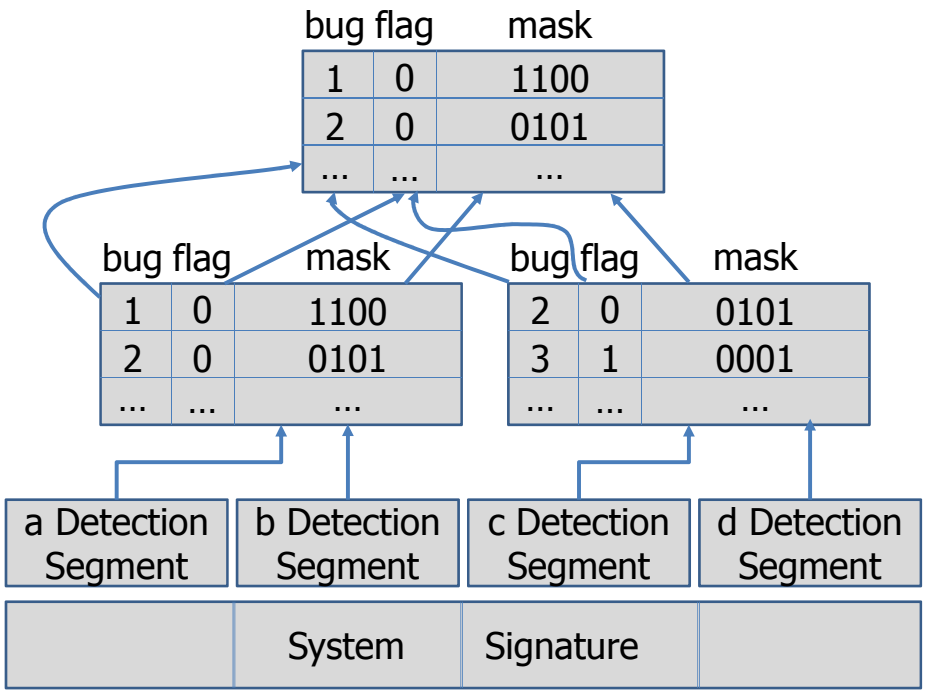


Figure 4.3: *Architecture of the processor bug detectors proposed by Constantinides* et al.*.*

overhead by reusing the pre-existing built-in self test hardware for bug checking. Once the signals are selected, circuitry is automatically added to the processor to support monitoring the selected signals' run time values against a system-wide bug signature. Circuitry is also added for storing multiple bug masks which encode which portion of the system signature each bug is sensitive to.

Signatures are representations of the current state of the system, *i.e.,* the concatenation of all values of signals that are flip-flop outputs. If we think of hardware as a large dataflow graph, starting from any bug, we can traverse the graph, stopping at flip-flops, and build the set of monitored signals that control a bug's activation. Thus, each bug has an associated signature which specifies the values the monitored signals must take on to activate the bug. Figure 4.4 shows a circuit that produces a buggy result and its associated signature. By monitoring the current state of the system and comparing it to each bug's signature, the bug detector knows when a bug is activated.

In most cases, there is no single, specific state of the processor associated

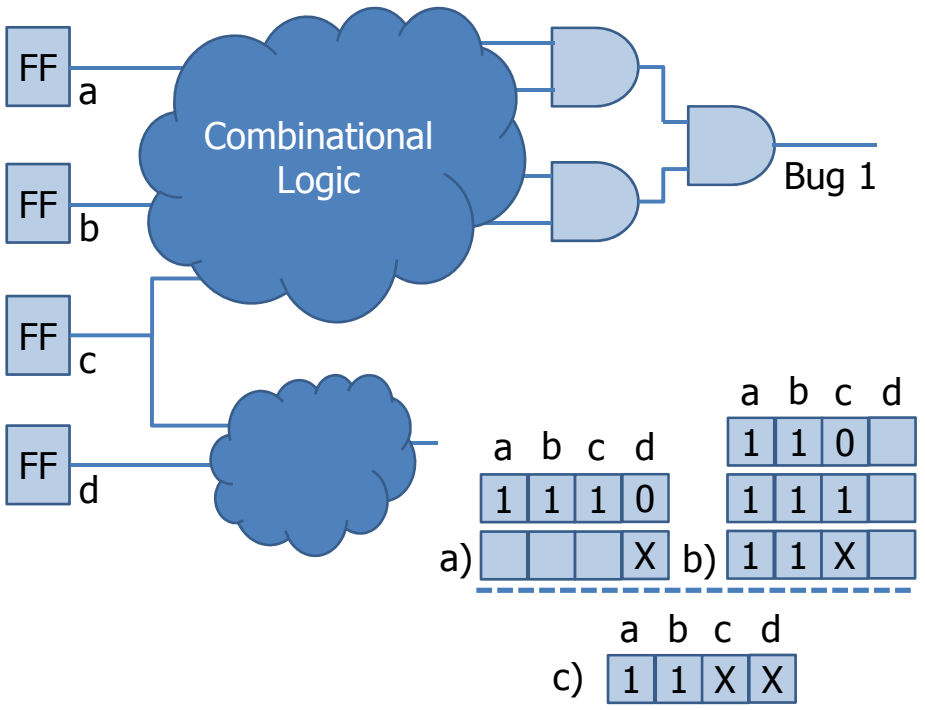


Figure 4.4: *Shows the creation of bug signatures and masks by using signals that are flip-flops outputs. Part a shows how the mask creates 'X' (don't care) values by encoding which segments can possibly affect the bug. Part b shows how 'X' values get added when multiple bug triggers are combined into a single signature. Part c shows the resulting bug signature which is a combination of the results of steps a and b.*

with a bug's activation. For instance, if there is no path connecting a signal 'A' and a buggy output 'B' in the dataflow graph representation of the hardware, then we can say that 'B' does not care what the value of 'A' is. This requires a third value possibility for each signature character–'X' (Don't Care). For each signal marked with an 'X' in a bug's signature, the detector can ignore the current value of the signal, because it does not contribute to the bug's activation. Figure 4.4(a) shows the addition of these types of 'X' values to a bug's signature.

It is also possible that multiple values of a monitored signal activate a bug. This case also produces an 'X' value for the conflicting bits, as shown in Figure 4.4(b). While the resulting 'X' value is the same as with the previous case, this situation can produce false detections. False detections occur when 'X' values accumulate inside a bug detection segment. The accumulated 'X'

values express more states of the system than the actual set of states that can activate the bug. False detections are not generally a problem with the previous case, as those 'X' values are not monitored in a bug detection segment for that bug, where in the later case they are.

4.3.2 Erratacator modifications

The primary modification Erratacator makes to the bug detector is connecting the bug detection signal to one of the processor's interrupts. This forces the processor to perform a pipeline flush in the event of a detected bug activation. Not just any interrupt will suffice. The interrupt used by Erratacator needs to be the highest priority resumable interrupt in the system. The interrupt can also never be disabled, which requires additional hardware support to avoid losing state when handling Erratacator exceptions that occur when all interrupts would normally be disabled. To prevent state loss in this situation, Erratacator adds detector-only copies of all exception registers and the stack register. This not only allows for the safe interruption of previously uninterruptable blocks of code, but also allows for recursive Erratacator exceptions.

It is also important that Erratacator exceptions do *not* spawn any other exception themselves. The most notable cases are exceptions dealing with the state of the memory hierarchy (*e.g.,* a page fault). To support this, Erratacator exceptions are only interruptable by other Erratacator exceptions. To avoid locking the processor by having a page fault, etc., occur when the firmware attempts to run, Erratacator firmware sits outside the traditional memory hierarchy of the processor.

As a byproduct of using the processor's interrupt mechanism, Erratacator exceptions are associated with an instruction, whether they are caused by an instruction or not [1]. It is critical for recovery that Erratacator associates the bug to an instruction that allows all contaminated state to be flushed out of the pipeline. All issued instructions before the trapped instruction are considered to have fully executed and have impacted the state of the system.

[1] As discussed in Section 4, many bugs are linked to a series of events, *e.g.,* an exception occurring on a page boundary, as opposed to being caused by a specific instruction. In this case, Erratacator associates the bug detection with the instruction preceding the exception and then simulates the effect of the exception

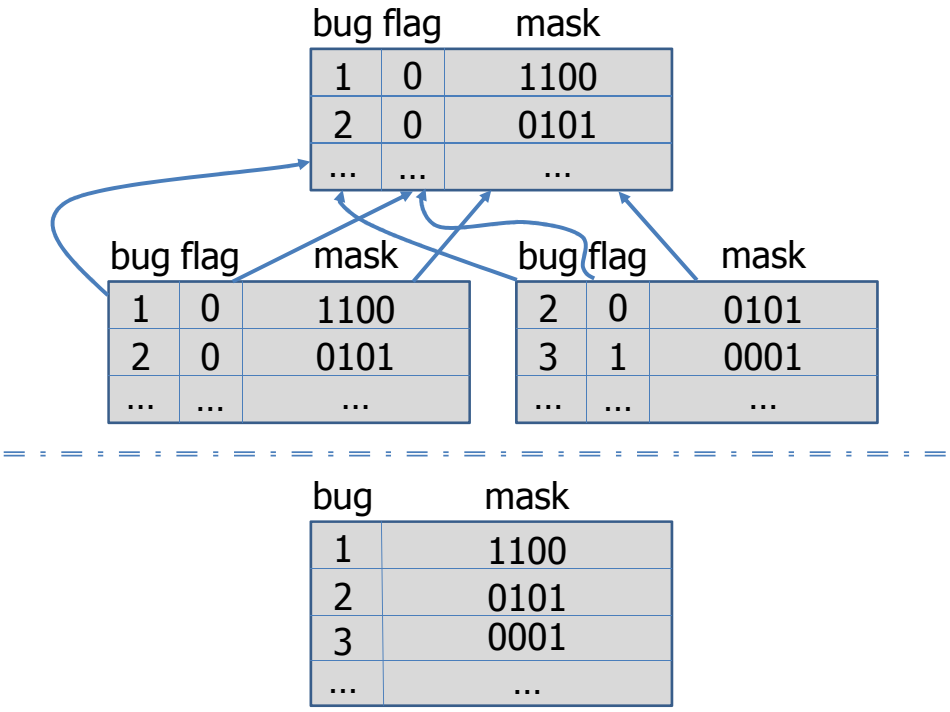


Figure 4.5: *Structural comparison of the segment checking tree (top) and its more compact and area efficient replacement, the bug table (bottom).*

To support correct bug/instruction associations, Erratacator adds an interrupt input to each stage of the pipeline before state changes are committed. This allows multi-level detection, *e.g.,* a bug that has detection segments composed only of signals in the write-back stage can cause an Erratacator exception to be attributed to the instruction currently in the fetch stage.

The last major difference between the bug detectors implemented in Erratacator and those proposed in Constantinides *et al.* is that Erratacator's detectors apply the signature generation and checking technique at the hardware definition language (HDL) level as opposed to the layout level. Since wire length is not a concern at the HDL level, we can combine all segment match detection tables into a single, unified bug table. Figure 4.5 shows a visual comparison of the two structures. The bug table also excludes the flag bit present in each row of the segment match detection tables. Condensing the tables into a single, more compact, bug table reduces the hardware area required by the detector in exchange for a more complicated layout, which the tools handle for us. Another benefit of moving to the HDL level over

Bug	Date	We Found	Link
1	05/2010		Waqas Ahmed's dissertation [79]
2	05/2010		Waqas Ahmed's dissertation [79]
3	05/2010		Waqas Ahmed's dissertation [79]
4	05/2010		Waqas Ahmed's dissertation [79]
5	05/2010		Waqas Ahmed's dissertation [79]
6	05/2010		Waqas Ahmed's dissertation [79]
7	10/2010		`http://opencores.org/bug,view,1787`
8	04/2011		`http://opencores.org/bug,view,1903`
9	04/2011		`http://lists.openrisc.net/pipermail/linux/2011-April/000000.html`
10	07/2011		`http://opencores.org/bug,view,1930`
11	07/2011		`http://bugzilla.opencores.org/show_bug.cgi?id=51`
12	10/2011	X	Not reported
13	11/2011	X	Not reported
14	12/2011	X	`http://lists.openrisc.net/pipermail/openrisc/2011-December/000531.html`
15	01/2012	X	Not reported
16	05/2012	X	`http://lists.openrisc.net/pipermail/openrisc/2012-May/001115.html`

Table 4.1: *A list of bugs that we implement, the date each bug was first reported, a link to a the report of the bug, and a column, "We Found", that contains a mark for each bug that we uncovered in the process of building Erratacator.*

the layout level is the replication of signals present at the HDL level. This replication provides more options for processor designers to describe a bug's signature, leading to fewer false detections.

4.4 The Erratacator prototype

To highlight the ability of Erratacator to overcome real-world processor bugs, we build a demonstration platform consisting of implementations of the bug detectors described in Section 4.3 and recovery firmware described Section 4.2. The demonstration platform is an OpenRISC OR1200-based

Bug	Type	Inst	M	Description
1	A		X	The overflow flag is not updated
2	A	X	X	The set of extend sub-word level instructions get executed as a l.MOVHI instruction
3	T	X		l.MULI does not reserve the appropriate number of cycles, leading to incorrect results
4	T			The carry flag is updated even when the pipeline is frozen
5	A			Division by zero does not set the carry bit
6	A	X	X	l.FL1 is executed as l.FF1
7	L	X	X	l.MACI instruction decoding incorrectly
8	A	X	X	l.MULU instruction not implemented, but does not throw an exception
9	T			l.RFE takes two cycles but may have only one to complete, corrupting data
10	T			l.MACRC preceded by any MAC instruction produces incorrect results
11	L			ALU comparisons where both a and b have their MSB set can produce incorrect results
12	T			Closely consecutive exceptions can cause the processor to stall
13	T			When an exception occurs shortly after an exception return, the Supervisor Register and Exception PC register get corrupted
14	L		X	Writes to register 0 are not ignored
15	A		X	Exception registers are not updated correctly for the align, illegal instruction, trap, and interrupt exceptions that occur in the delay slot
16	T			When the MMU is first enabled, the processor prioritizes speculative data from the bus using the virtual address as the physical address as opposed to using cached data from the translated address

Table 4.2: *Details about each processor bug that we implement. Column "Type" lists the classification for each processor bug using a taxonomy which labels bugs as either (L)ogic, (T)iming, or (A)lgorithm [9]. Column "Inst" contains marks for the processor bugs that are directly tied to instructions, thus suitable for patching using conventional mechanisms such as microcode [84, 59], re-compilation [40], and translation [85]. The "M" column marks bugs which are due to missing functionality: as opposed to errata-like functionality, which we target. Finally, we provide a description of each bug.*

System-on-Chip [2] [86] with 16 publicly documented bugs [79, 80, 81]. Table 4.1 shows the date and a pointer to the documentation for each of the 16 bugs that we implement in Erratacator. We provide a description of the bug sufficient for understanding the evaluation results and highlight some other important properties of each bug that we implement in Table 4.2. The most important bug property listed in Table 4.2 is the maturity of a bug: our work is focused on errata-like bugs, similar to what you would find in Intel's specification updates.

The OpenRISC OR1200 processor is an open source, 32-bit, RISC processor with a five stage pipeline. The processor includes 32KB instruction and data caches and MMU support for virtual memory with 64 entry instruction and data translation look-aside buffers. Commercial products using the OR1200 include Jennic Ltd.'s Zigbee RF Transceiver and Beyond Semiconductors BA series of chips. The OR1200 will soon make its way into space as a part of NASA's TechEdSat program [87].

The reason for selecting the OR1200 is that both the processor and the bugs are open source, which we found not to be the case with other open source processors (*e.g.,* Leon3 [88] and OpenSPARC [89]). In all, we found 50 unique bugs spanning the six years that the processor has been in development (1 bug every 45 days). Due to time constraints, we implement 16 of the 50 bugs, basing our selection on diversity and which bugs were available through SVN rollbacks. By combining the public documentation with previous versions of the processor in its SVN repository, we are able to accurately reproduce the processor bugs used in our implementation.

[2]We couple the OR1200 with several other hardware units to form a system capable of loading large programs and communicating with the user. This includes JTAG debug hardware, 256 MB of DDR2 memory, 10/100 Ethernet, and a UART.

Table 4.3: *The signature and mask for each bug in our implementation. Bugs that require more than one signature and have a alphabetic postscript after thier ID. By default, all bits of every signal in a bug's signature are not masked. But bugs with multiple signatures may have conflicting signal values it the signature. We mask such bits off and treat them as always matches. The "Mask" column lists the masked bits signals with conflicting signature valuess; zero bits denote a masked bit. *** denotes where we moved detection earlier in the pipeline to avoid inter-bug signal contention. +++ signifies that the detector overpredicts due to the limited expressiveness of the processor's registers. ### denotes where we made the signature less precise to reduce the possibility of inter-bug signal contention in the system mask.*

Bug	Signature	Mask
bug1a bug1b	ovforw = 1, ov_we_alu = 1 ovforw_mult_mac = 1, ov_we_mult_mac = 1	
bug2	id_insn[31:26] = 111000, id_insn[4:0] = 01100	
bug3	id_insn[31:26] = 101100, id_insn[9:8] != 11	
bug4	if_insn[31:26] = 111000, if_insn[4:0] = 00001, if_pc[3:0] = 1000	***
bug5a bug5b bug5c bug5d	alu_op = 01010, div_by_zero = 1 alu_op = 01010, a = 0x80000000, b = 0xffffffff alu_op = 01001, div_by_zero = 1 alu_op = 01001, a = 0x80000000, b = 0xffff_ffff	alu_op = 11100
bug6	id_insn[31:26] = 111000, id_insn[4:0] = 01111, id_insn[9:8] = 01	
bug7a bug7b bug7c bug7d	alu_op = 00110, ex_freeze_r = 0 alu_op = 01011, ex_freeze_r = 0 alu_op = 01001, ex_freeze_r = 0 alu_op = 01010, ex_freeze_r = 0	alu_op = 10000
bug8	id_insn[31:26] = 111000, id_insn[4:0] = 01011	

Continued on the next page

Table 4.3 – continued from previous page

Bug	Signature	Mask
bug9	ex_freeze = 0, ex_branch_op = 110, sr[6] = 0	+++
bug10a	id_insn[31:26] = 000110, id_insn[16:0] = 0x10000, ex_insn[31:26] = 010011, ex_insn[3:2] = 00	ex_insn[31:26] = 011101
bug10b	id_insn[31:26] = 000110, id_insn[16:0] = 0x10000, ex_insn[31:26] = 110001, ex_insn[3:2] = 00	
bug11a	comp_op[3] = 0, sum[31] = 1, a[31] = 0, flag_we_alu = 1	
bug11b	comp_op[3] = 0, sum[31] = 1, b[31] = 1, flag_we_alu = 1	
bug12a	ex_insn[31:26] = 000101, ex_insn[16] = 1, state = 000, sig_int = 1	
bug12b	ex_insn[31:26] = 000101, ex_insn[16] = 1, state = 000, sig_tick = 1	
bug13a	wb_branch_op = 110, sig_int = 1	
bug13b	wb_branch_op = 110, sig_tick = 1	
bug14	rf_addrw = 00000	###
bug15	if_pc[31:0] = 0x504, id_pc[31:0] = 0x600, ex_pc[31:0] = 0x700, if_pc[31:0] = 0x804	*** if_pc[31:0]=0xFFFFF2FF
bug16a	state = 001, hitmiss_eval = 1, tagcomp_miss = 0, dcqmem_ci_i = 0, load, cache_inhibit = 1, biudata_valid = 1	

Continued on the next page

Table 4.3 – continued from previous page

Bug	Signature	Mask
bug16b	state = 001, dcram_we_after_line_load = 1, load = 1, cache_inhibit = 1, biudata_valid = 1	

We implement the Erratacator prototype on the Digilent XUPV5 development board [90]. Table 4.3 shows the signature and mask for each bug in our implementation. The SoC runs with a maximum frequency of 90 MHz, consuming 4,195 of the Virtex 5 FPGA's 17,280 LUTs, 24%, and 28 of its 148 Block RAMs, 19%. In an ASIC implementation, the target for Erratacator due to its static hardware configuration, Constantinides *et al.* [9] propose reusing scan chain resources—keeping with the theme of reusing existing processor resources to eliminate overhead and increase performance. This reduces hardware area overhead to as little as 10%, to which we add four 32-bit registers to support the ISA extension.

The firmware portion of Erratacator comprises 4456 lines of C code (including formal markup) and 163 lines of assembly code compiled with GCC and the "-O3" option. The resulting binary is 14476 bytes which runs from system memory.

Erratacator successfully runs all benchmarks on both Linux and bare metal, with all experiments running on Linux 3.1.

4.5 Erratacator evaluation

The goal of these experiments is to answer the key questions about Erratacator's performance and scaling: Does Erratacator actually work, what is the cost of protection, and how does the cost of protection scale with the number of bugs monitored. Motivated by poor scaling, we explore the impact of exposing the reliability/performance tradeoff to trusted software. Finally, we explore the relationship between the rate of recovery firmware activations and the run time overhead experienced by software.

For all experiments, we use MiBench [91], a set of embedded computing benchmarks that stress different aspects of the processor. MiBench offers benchmarks from six general areas: automotive, consumer, office, network, security, and telecommunications. We include one application from each area.

All experiments are run on the platform described in Section 4.4, except we run the processor at 50 MHz. A lower frequency makes building different hardware configurations quicker; resulting in increased researcher productivity. Also, due to the poor heat dissipation of the development board, running at the the maximum frequency eventually overheats the FPGA, shortening its life and causing transient faults.

4.5.1 Does Erratacator detect and safely recover from processor bugs?

Bug	Detect	Recover	Signals	False Positives
1	√	√	4	
2	√	√	2	
3	√	√	3	
4	√	√	3	√
5	√	√	4	√
6	√	√	3	
7	√	√	2	√
8	√	√	2	
9	√	√	2	√
10	√	√	4	√
11	√	√	3	
12	√		5	
13	√	√	3	
14	√	√	2	√
15	√		1	
16	√	√	3	

Table 4.4: *Erratacator's ability to detect bugs, recover from bugs, the number of signals monitored by each bug's detector, and if the detector for each bug can produce false detections, which then create unneeded activations of the recovery firmware.*

The primary goal of Erratacator is to prevent software state contamination due to processor bugs. Therefore, the most important question is whether

Erratacator can detect and flush the effects of the processor bug before it contaminates ISA-level state, and upon detection, if Erratacator's recovery firmware allows the software to make correct forward progress.

For this experiment, we implement 16 of the publicly documented OR1200 bugs, 5 of which we found while implementing Erratacator (patches accepted up stream). Table 4.2 provides details of each bug. For each bug, we implement it, a software-level test case that activates the bug, and a signature and mask for the detection fabric. We use the test case to ensure that the detector detects bug activations and then maintains a consistent state. Note that the software-level test case is not meant to activate a bug in every way possible as verifying bug signatures is not the focus of our work.

Table 4.4 shows the results of this experiment, including the number of hardware level signals used in each bug's signature and whether the signature could produce false detections on its own. The takeaway is that Erratacator successfully detects all 16 bugs. Erratacator also maintains a consistent ISA-level state, pushing software state forward safely, for 14 of the bugs.

The two bugs that Erratacator could not recover from involve flaws in the processor's exception processing facilities. bug12, is not recoverable in cases of unpredictable exceptions (*i.e.,* interrupts) because it involves the processor corrupting micro-architectural state when it attempts to process an new exception in the delay cycles after starting to process a previous exception. In such cases, the processor deadlocks due to the corrupted values. bug15, unlike bug12, is generally recoverable using the techniques presented in this dissertation, but not recoverable in this implementation as Erratacator repurposes the one of the affected exceptions.

bug4 presents another interesting issue, the ability to tradeoff earlier bug detection for a possible increase in false detections. bug4 causes the supervisor register to update the carry flag irrespective of any pipeline events (*e.g.,* a stall). In the first implementation, by the time bug2 was detected, the inconsistent state was committed. By adjusting the detectors to detect the bug one pipeline stage earlier, Erratacator is able to correctly flush all contaminated state. Early detection of bugs is also a tool for balancing contention among bug signatures on the limited system signature and system mask resources, reducing false detections.

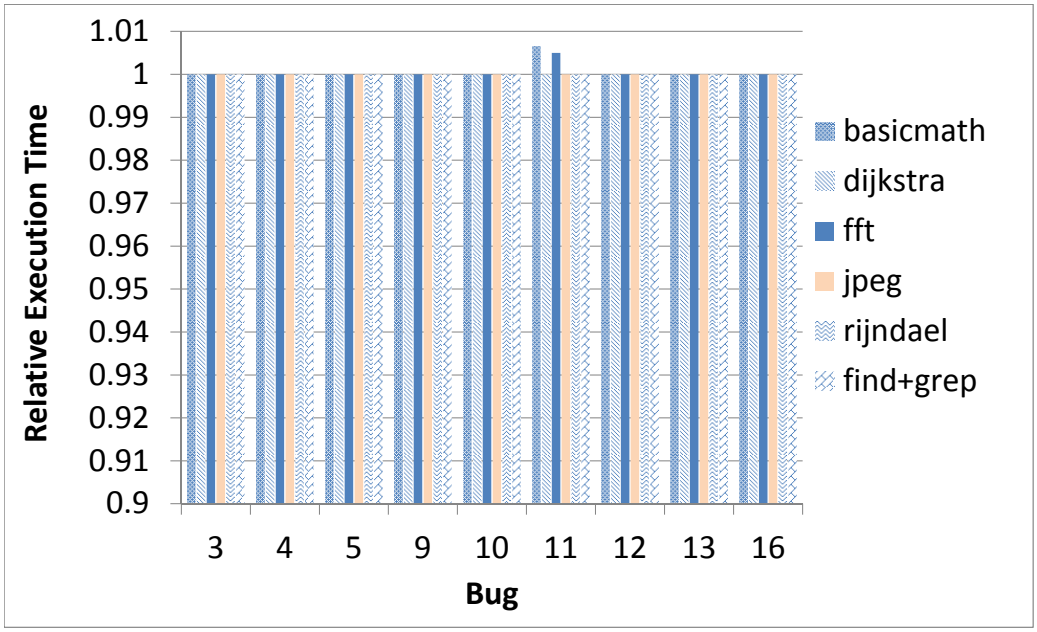


Figure 4.6: *Relative execution time for each bug and benchmark*

4.5.2 Does Erratacator scale with the number of bugs monitored?

The goal of the first experiment in this section is to determine the run time overheads due to Erratacator recovering from mostly true detections of single bug activations. The second experiment looks at the effects of Erratacator monitoring multiple bugs. In both experiments, the set of bugs used contains only those bugs which are representative of errata found in commercial processors, *i.e.,* those not marked in the "M" column in Table 4.2. Bugs due to processor immaturity (*i.e.,* "we haven't implemented overflow yet") showed a much higher activation rate and signature collision rate, and are our not focus since commercial ASIC processor don't have such issues.

In the first experiment, we look at the behavior of Erratacator when it monitors a single bug at a time. Figure 4.6 shows the software run time overhead for this lone bug experiment. The general lack of bug detections shows—as expected or errata—that most bugs are so hidden in the processor's state space that benchmarks almost never activate them. bug11 is the exception. Both the basicmath and fft benchmarks activate bug11 by containing instructions that compare two negative numbers, behavior expected in math heavy benchmarks. These two benchmarks trigger bug11 between 80

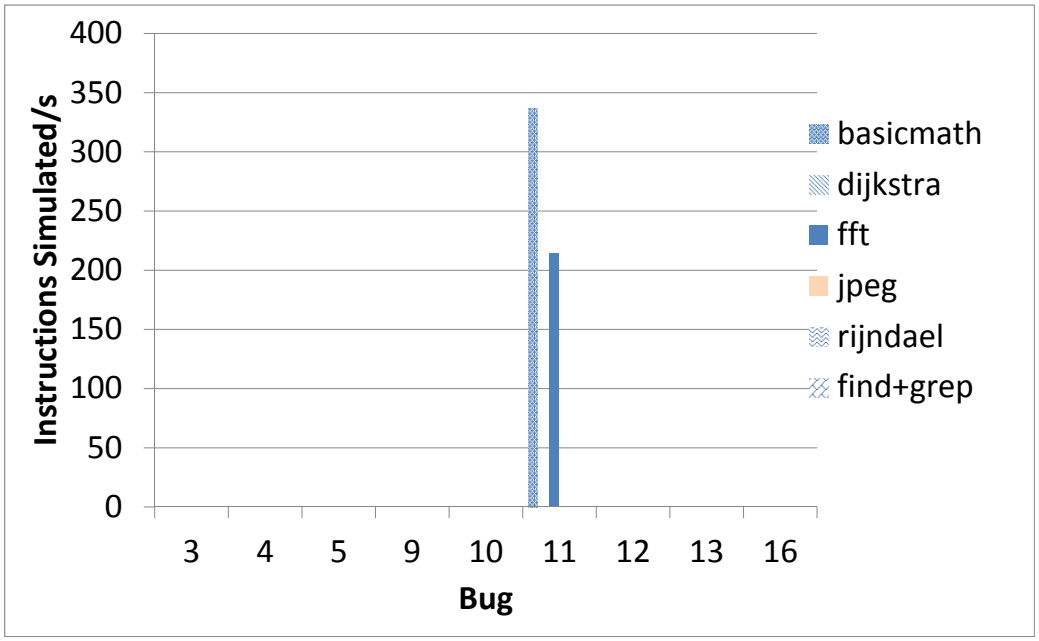


Figure 4.7: *Number of benchmark instructions per second Erratacator recovery firmware simulates for each bug*

and 120 times a second on average. Figure 4.7 shows how this bug detection frequency translates into the number of instructions per second that Erratacator must simulate on behalf of software. Infrequent activations coupled with efficient recovery produces software run time overheads of less than 1%.

Monitoring a lone bug is not sufficient. Since commercial processors have upwards of one hundred bugs, and the ideal case is to avoid all bugs, all of the time, the next experiment seeks to explore how software run time overheads scale with the number of bugs monitored. Using the same platform as in the previous experiment, in this experiment, instead of monitoring only one of the nine errata-like bugs, Erratacator monitors an additional bug at each iteration, with bugs added in the order of discovery.

Figure 4.8 shows the number of bug detections per second due to Erratacator monitoring the ever-increasing number of bugs. The figure shows that adding bug10 dramatically increases the rate of bug detections—up to 4.5 million a second. At first glance, this seems odd, especially since none of the bugs up to and including bug10 are triggered by any of the benchmarks (Figure 4.7). These are false detections resulting from the signatures and masks of bugs 3, 4, 5, 9, and 10 competing for the same, limited, resources. Since the system signature and mask contains one bit for each flip-flop in

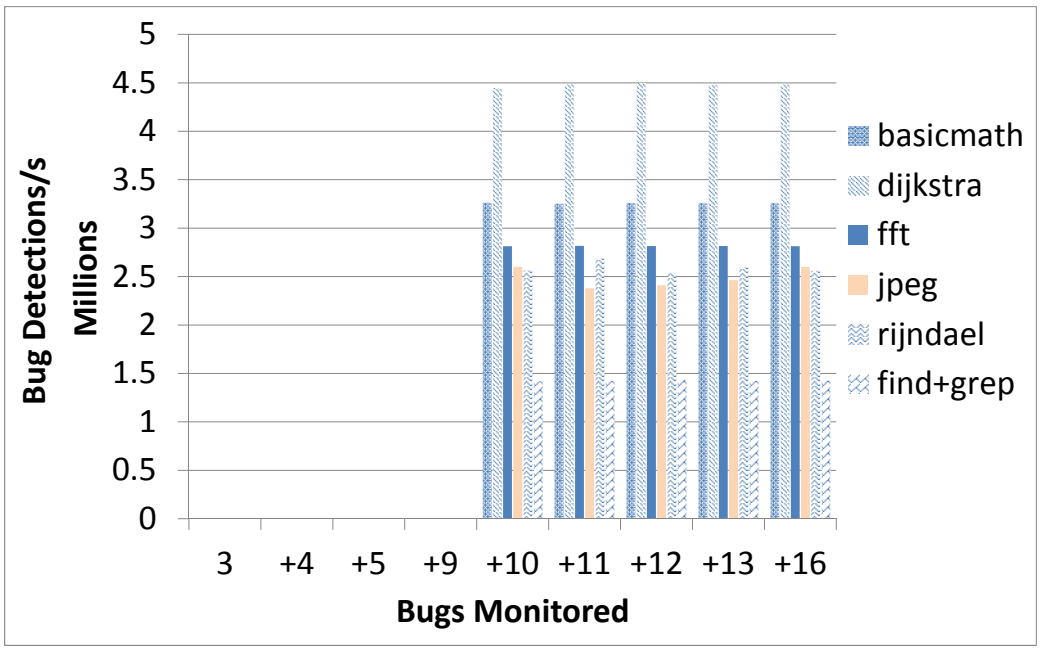


Figure 4.8: *Number of Erratacator exceptions per second experienced during each benchmark as more bugs are monitored*

the hardware, bugs that require the access to the same flip-flop values must have their signature and mask merged. Merging in a way that avoids missing bug activations leads to false detections. In the case of this experiment, the contention that causes the spike in detections comes from opcode bit competition. This causes bugs that are heavily tied to specific instructions (*e.g.,* 3, 4, and 10) to fire for almost every instruction, with the differences in rates being similar to the instruction commit rate for each benchmark.

To address the problem of over-contention for the limited system signature and mask resources, we add an ISA-level interface to Erratacator that empowers trusted software with the ability to manage its own reliability and performance. By controlling which bugs are monitored, software can control bug fabric resource contention, thus controlling the likelihood of false detections. To gain insight into the affect of software managed reliability, we conducted an experiment where software could, in effect, disable bug detectors for bugs it could guarantee would never be activated. A prime candidate for disabling was bug10, a bug in the multiply accumulate (MAC) logic, because the kernel and benchmarks never use any MAC instructions. If a program were to use a MAC instruction, the kernel could always enable bug10.

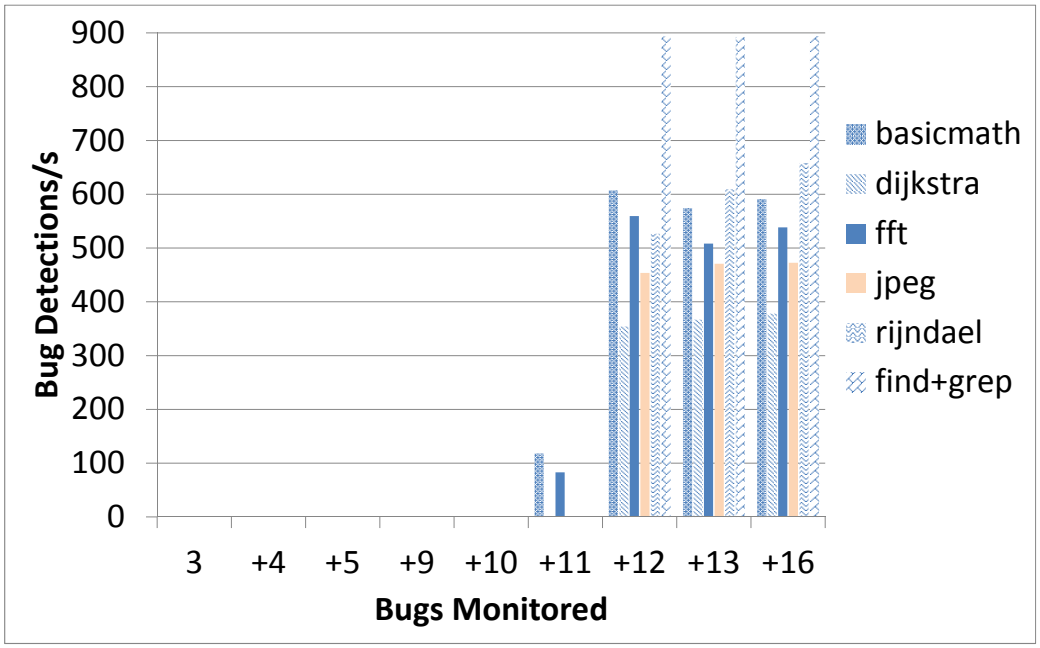


Figure 4.9: *Number of Erratacator exceptions per second experienced during each benchmark as more bugs are monitored with software in control of its own reliability*

Figure 4.9 shows the impact of disabling a bug10 when it is guaranteed to never activate. There is no overhead until bug11 is added, which coincides with the fact that none of the previous bugs had activations in the lone bug experiment and none of the bugs previous to bug11 have signature conflicts with each other.

4.5.3 What is the cost of recovery?

The last experiment looks at how the run time overhead due to Erratacator recovery is related to the frequency of bug detections. To do this, we modify Erratacator to have a countdown timer trigger exception instead of a bug detection triggered exception. After Erratacator's recovery firmware handles the exception, the countdown timer resets to a re-programmable value. This way, testing software can sweep through a range of counter values, controlling the frequency of recovery firmware invocations. Because the overhead in this configuration is relatively in dependent of the instruction stream, the six benchmarks from previous experiments are combined with only a single cumulative time displayed.

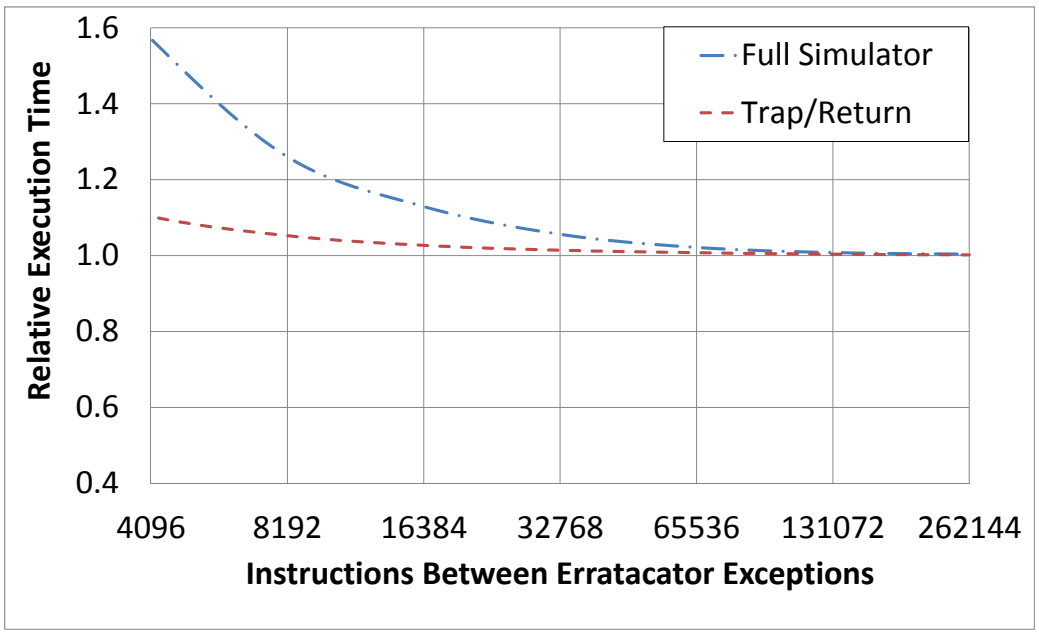


Figure 4.10: *The top line (large dashes) represents the overhead when Erratacator recovery firmware runs with full simulation. The bottom line (small dashes) represents the overhead due to taking an exception, backing-up the software state, and then restoring it,* i.e., *trapping then returning. Six benchmarks combined.*

Figure 4.10 shows the results of this experiment, sweeping between 4096 and 262,144 instructions between recovery firmware invocations. There are a range of recovery options from just taking the exception and returning to full simulation. The two options shown represent opposing ends of a tradeoff with trap/return being lower overhead, but less likely to recover and full simulation being more overhead, but more likely to recover.

Results indicate that there is a linear relationship between the number of recovery firmware invocations and execution times. The graph also shows that as the frequency of invocations increases, the difference between just trapping and returning and employing the full firmware grows. This highlights an opportunity for using more time consuming recovery routines in an effort to reduce the likelihood of future bug detections.

4.5.4 Why recovery works

In this section, we describe why recovery works for each errata-like bug in our implementation. For each bug, we show a snippet of code from the

Bug	Type	Description	Why recovers
3	T	l.MULI followed by a result read	Trap and return
4	T	The carry flag is updated prematurely	Restricted environment
9	T	l.RFE returns to the wrong address	Trap and return
10	T	l.MACRC reads incomplete results	Trap and return
12	T	Processor stall due to consecutive exceptions	No recovery
13	T	Bad updates to exception registers	Recoding
16	T	Wrong data sourced used for a load	Trap and return
11	L	Bad ALU comparisons	Recoding
5	A	Division by zero does not set carry	Recoding

Table 4.5: *Why recovery works. Column "Type" lists the classification for each processor bug using a taxonomy which labels bugs as either (L)ogic, (T)iming, or (A)lgorithm [9]. We also provide a description of each bug and a high-level idea of why recovery works.*

OR1200 processor that contains the bug (commented out) and the fix. Using the code snippet for reference, we describe how software activates the bug and the possible effects that the activated bug can have on software. With an understanding of how software activates the bug and the bug's effects on software, we describe the specific set of system states and events that differentiate the buggy line of code from the correct line of code. Knowing which system states and events to avoid leads naturally to a discussion of how our recovery mechanism recodes the instruction stream to avoid reactivating the bug. Before going into detail on how Erratacator recovers from each bug, we summarize the results of the analysis and discuss the high-level takeaways.

Summary

Table 4.5 provides a high-level summary of why recovery works for each errata-like bug in our implementation. The table organizes bugs by their type (according to the taxonomy proposed by Constantinides *et al.* [9]), with

the aim of seeing if similar bug types have similar recovery strategies. As shown in the table, we found three classes of recovery: (1) Trap and return; (2) Restricted environment; and (3) Recoding.

Trap and return recovery is when taking an exception and returning immediately from the interrupt service routine perturbs the processor enough to allow software execution past the bug. Trap and return recovery is sufficient for recovering from bug3, bug9, bug10, and bug16. Notice that all of these bugs are related to the timing of processor states and events.

Restricted environment recovery is when executing a potentially bug activating instruction in the more restrictive environment of the recovery firmware (in the sense that there are less events that can occur) is guaranteed to avoid activating the bug. One example of how executing inside the recovery firmware restricts the environment is the lack of exceptions. Instructions running in the recovery firmware will never be interrupted because the recovery firmware runs as the highest priority code on the system and during its execution, the processor ignores all other exceptions and interrupts. bug4 represents a case where a restricted environment is sufficient for recovery.

The last class of recovery is the recoding scheme that we discuss in Section 4.2. In this class of recovery, we recode instruction streams from the software stack to route execution around the bug. This is the most general-purpose recovery class of the three as it encompasses the other two classes, but it is also the most costly in terms of software run time overhead. bug13, bug11, and bug5 require this level of recovery.

bug3

This bug is the result of the processor not reserving enough cycles for the l.MULI instruction. Listing 4.6 shows the source code for the bug and what the fix looks like. The purpose of the code in the listing is to determine (during the decode stage) how many cycles to freeze the pipeline for in the execute stage. It looks like the processor's designers copied the code for the l.MACRC instruction. The problem is that l.MULI uses an immediate operand, which is at bit index 16, while l.MACRC does not have an immediate operand (bit 16 is a reserved bit in l.MACRC).

So, the processor only reserves enough time for l.MULI to execute when the value its immediate has a 1 in index 16. Even when the bit at index 16

```
// Encode wait_on signal
always @(id_insn) begin
   case (id_insn[31:26])
     `OR1200_OR32_ALU:
       wait_on = ( 1'b0
     `OR1200_OR32_MULI:
       // BUG: 17th bit of instruction is immediate
       // wait_on = id_insn[16] ? `OR1200_WAIT_ON_MULTMAC :
       // `OR1200_WAIT_ON_NOTHING;
       // FIX: stall the pipeline for the needed cycles
       wait_on = `OR1200_WAIT_ON_MULTMAC;
     `OR1200_OR32_MACRC:
       wait_on = id_insn[16] ? `OR1200_WAIT_ON_MULTMAC :
                               `OR1200_WAIT_ON_NOTHING;
```

Listing 4.6: *bug3 source code*

is a 0, software may still get correct results as the multiply unit continues to process in the background. Software will only see incorrect results when the bit at index 16 is 0 and a later instruction uses the output of l.MULI before the multiplier completes execution. In that case, the instruction that uses the result of l.MULI sees partial results, which then corrupts its own result.

In this case, we recover by executing a series of shifts and adds in the place of l.MULI. These instructions complete in a single cycle and completely avoid activating the buggy line shown in Listing 4.6, thus routing execution around the bug.

bug4

This bug is the result of the carry flag in the supervisor register being updated regardless of any pipeline freezes or pipeline flushes. Listing 4.7 shows that the correct line includes information on the state of the pipeline (*i.e.,* if it is frozen or flushed), but the buggy line does not. This means that the most recent carry result is always the one stored—and available to future instructions—in the supervisor register, even if the instruction that created the carry value is frozen in the execute stage or has been flushed. This means later instructions will potentially see a carry value not consistent with the rest of the system's state, creating an off-by-one error. For instance, if a program issues a l.ADDC instruction with the carry flag set, but where it

```
// Register file write enable is either from SPRS or normal from CPU control
always @(`OR1200_RST_EVENT rst or posedge clk)
        if (rst == `OR1200_RST_VALUE)
                rf_we_allow <= 1'b1;
        else if (~wb_freeze)
                rf_we_allow <= ~flushpipe;

assign rf_we = ((spr_valid & spr_write) | (we & ~wb_freeze)) & rf_we_allow;

// BUG: not taking into account freezes of flushes
// assign cy_we_o = cy_we_i;
//FIX: Only enable writes when not frozen and not flushed
assign cy_we_o = cy_we_i && ~wb_freeze && rf_we_allow;
```

Listing 4.7: *bug4 source code*

does not produce a carry, and an exception occurs, when the instruction is executed again, there will be no carry and the result will be incorrect.

The critical difference between the correct line and the buggy line shown in Listing 4.7 occurs when the instruction is flushed from the pipeline, *i.e.,* the instruction's results are not committed to ISA-level state, and it gets re-executed again. As long as the recoded instruction sequence does not exhibit this specific behavior, recovery will be successful. This is the cases for execution inside the recovery firmware. Erratacator runs as the highest priority code on the system and it is not interruptable by any other exception. This means that it is impossible to interrupt a carry using instruction when it is executed inside the recovery firmware, meaning no risk of pipeline flushes invalidating carry results prematurely stored to ISA-level state. In this case we do not even need to recode the instruction, just change slightly the environment that it executes in.

There is still the problem of meeting the recovery firmware's assumptions with this bug. The recovery firmware assumes that the ISA-level state is consistent when it starts execution, but this bug updates—potentially corrupting—ISA-level state after one cycle in the execute stage—even if it is frozen. This requires that we detect the bug early, in the decode stage, and recode and execute both the bug inducing instruction and the instruction before the bug inducing instruction.

```
assign alu_op_sdiv = (alu_op == `OR1200_ALUOP_DIV);
assign alu_op_udiv = (alu_op == `OR1200_ALUOP_DIVU);
assign alu_op_div = alu_op_sdiv | alu_op_udiv;

// If b, the divisor, is zero and this is a division
assign div_by_zero = !(|b) & alu_op_div;

// BUG: No assignment to the carry flag
// FIX: Drive the carry flag
assign cy_div = div_by_zero;
```

Listing 4.8: *bug5 source code*

bug5

The OR1000 architecture specification states that division by zero results in the processor setting the carry flag. The section of code in Listing 4.8 shows that the processor does not assign anything to the carry flag based on the results of division. Thus, when software performs a divide-by-zero, the carry flag will not change and software will have no way of knowing of the divide-by-zero condition. Note that all other division operations produce correct results as the processor specification states that division operations only change the value of the carry flag in divide-by-zero situations.

Therefore, for recovery we just need to make sure that we set the carry flag in divide-by-zero conditions. A valid recovery routine in this case consists of executing the division inside the simulator—no recoding—and checking if the divisor is zero, updating the carry flag if it is. The actual recovery routine that we employ is more complicated, but also more general purpose. We recode the division into a set of shift, subtraction, AND, and OR operations, also including a check for zero valued divisors. Our general-purpose recoding routine incurs more overhead, but is likely to recover from a greater number of bugs in the division unit.

bug9

Listing 4.9 shows a bug where the processor does not treat the return from exception instruction (l.RFE) as a multi-cycle instruction. This bug produces an error when software returns from an exception and the address that the processor's exception handling logic passes control to (stored in the Exception

```
// Decode of multicycle
always @(id_insn) begin
  case (id_insn[31:26])
    // Return from exception
    `OR1200_OR32_RFE,
      // BUG: l.RFE does not reserve two cycles
      // multicycle = `OR1200_ONE_CYCLE;
      // FIX: request two cycles when you see l.RFE
      multicycle = `OR1200_TWO_CYCLES;

    `OR1200_OR32_MFSPR:
      // to read from ITLB/DTLB (sync RAMs)
      multicycle = `OR1200_TWO_CYCLES;
    // Single cycle instructions
    default: begin
      multicycle = `OR1200_ONE_CYCLE;
    end
  endcase
end
```

Listing 4.9: *bug9 source code*

Program Counter Register, EPCR) is resident in the instruction cache. When this situation occurs, the Instruction MMU will not have enough time to perform the address translation before the cache acknowledges the request (physically addressed caches). When software encounters this situation, it receives corrupt data from the instruction cache and executes the wrong instruction.

The system conditions where the difference between the correct line and the buggy line (shown in Listing 4.9) become meaningful are when the l.RFE instruction requires two cycles to determine the correct return address, but it only gets a single cycle to execute. The specific case when l.RFE requires two cycles to execute is when there is a I-TLB miss, but an I-Cache hit. The specific case where l.RFE only gets a single cycle to execute is when there is no freeze in the execute stage of the pipeline: it moves straight to the write-back stage.

An ad hoc recovery routine could simply read the return address from EPCR, invalidate the cache line associated with the return address, and return to the software stack. The software stack will then be able to execute the l.RFE instruction with no risk of hitting the bug—because there will be

```
// BUG: The signal mac_stall_r didn't exist and MAC operations
// would conflict with a proceeding l.MACRC
// FIX: Add a stall signal to freeze the pipeline while l.MACRC completes
// Stall CPU if l.macrc is in ID and MAC still has to process l.mac
// instructions in EX stage (e.g. inside multiplier)
// This stall signal is also used by the divider.
always @('OR1200_RST_EVENT rst or posedge clk)
  if (rst == 'OR1200_RST_VALUE)
    mac_stall_r <= 1'b0;
  else
    mac_stall_r <= (|mac_op | (|mac_op_r1) | (|mac_op_r2)) &
                   (id_macrc_op | mac_stall_r);
```

Listing 4.10: *bug10 source code*

a cache miss. An even simpler, but more risky, recovery strategy would be to just take the trap and return to the software stack, relying on the I-TLB to load the translation for the return address.

Our general-purpose recovery routine, on the other hand, executes the l.RFE instruction inside the simulator, using the virtualized state of software. The simulator computes directly the return address of the instruction and transfers control only in its virtual state. Also, there is no risk of the simulator activating this bug itself when it returns control to the software stack using the l.REF instruction, because it disables the cache as part of its entry routine.

bug10

The multiply-accumulate (MAC) unit has it own pipeline of instructions so that the core pipeline of the processor does not have to wait for multi-cycle MAC instructions to complete before advancing. The only time the processor needs to wait for the completion of a MAC instruction is when software queries its value, which is rare because, as opposed to traditional instructions, most MAC instructions do not expect a return value after executing. On the OR1200, the only MAC instruction that can read the value of the MAC unit is the MAC read and clear instruction (l.MACRC).

As shown in Listing 4.10, bug10 is the result of a the l.MACRC instruction not waiting for the MAC pipeline to empty before storing the current accumulator value in the instruction's destination register. This bug manifests

```
assign {cy_sum, result_sum} = (a − b + 1) + carry_in;

// signed compare when comp_op[3] is set
assign a_lt_b = comp_op[3] ? ((a[width−1] & !b[width−1]) |
                (!a[width−1] & !b[width−1] & result_sum[width−1])|
                (a[width−1] & b[width−1] & result_sum[width−1])):
                // BUG 1: Have no look at all bits
                // a < b if (a − b) subtraction wrapped and a[width−1] wasn't set
                // (result_sum[width−1] & !a[width−1]) |
                // if (a − b) wrapped and both a[width−1] and b[width−1] were set
                // (result_sum[width−1] & a[width−1] & b[width−1] );
                (a < b);
```

Listing 4.11: *bug11 source code*

as an error when a l.MACRC instruction is preceded by any other MAC instruction. Software that triggers this bug will see stale results from the accumulator.

This, like the previous bug, is a case where recovery is achieved by taking the trap created by the bug detectors and returning control back to the software stack. Doing this creates enough of a time buffer between earlier MAC instructions and the l.MACRC instruction that there is no risk of the MAC pipeline being active when the processor executes the l.MACRC instruction. Our recovery mechanism works in much the same manner, but our time buffer comes from the need to fetch, decode, and recode another instruction from the software stack: we only recode one instruction at a time.

bug11

The bug shown in Listing 4.11 has taken many forms in the life of the OR1200 processor. The code listing shows the processor designer's final attempt to fix the issue before punting to the synthesis tool to implement the comparison correctly. Generally speaking, the bug involves an incorrect comparison of large (*i.e.,* the two most significant bits are set) unsigned operands. An example set of values (kept to 4-bits for clarity) that trigger this bug are a = 0111 and b = 0110: a > b. Given those values, result_sum = 1101 and the processor incorrectly reports that a < b. Software that attempts to compare two sufficiently large numbers will get results that incorrectly favor operand b being larger than operand a. These corrupted comparisons not

```
except_type <= `OR1200_EXCEPT_ILLEGAL;
epcr <= ex_dslot ?
        wb_pc : delayed1_ex_dslot ?
        // BUG: Wrong PC saved due to delay slot
        // id_pc : delayed2_ex_dslot ?
        // id_pc : id_pc;
        // FIX: Keep track of two levels of delay
        dl_pc : delayed2_ex_dslot ?
        id_pc : ex_pc;
```

Listing 4.12: *bug15 source code*

only contaminate data, but lead to incorrect control flows.

To recover from buggy comparisons, the recovery firmware recodes comparisons using other comparison instructions and by flipping the operands. In this case, to route execution around the buggy less than comparison (a < b), the recovery firmware will compute (a >= b), then invert the result, finally updating the virtualized flag register. In cases where all compare operations rely on the same hardware-level compare logic, it is possible to adjust both operand values, or even use arithmetic operations and check the carry flag.

bug12

We cannot recover from this bug, because the bug creates errors in how the processor handles exceptions and the detectors heavily rely on the processor's exception handling. Specifically, this bug causes the processor to lock in the exception handling finite state machine.

bug15

This bug is a failure of the processor's exception handling mechanism to correctly save the Program Counter (PC) in the event of certain exceptions. Ideally, when an exception occurs, the processor atomically saves the state of software that gets overwritten when the processor passes control to the exception handler. In the case of the OR1200, the instruction set specification dictates that the processor saves the PC in the Exception Program Counter Register (EPCR), the Supervisor Register (SR) in the Exception Supervi-

sor Register (ESR), and for instructions that compute an address (*e.g.,* load and store), the effective address in the Exception Effective Address Register (EEAR). Saving the correct PC value is not straight forward in the OR1200 due to the delay slot instruction: the instruction after a branch/jump that always gets executed. If an exception occurs in a delay slot, the processor must save the PC of the previous (branch/jump) instruction in the EPCR. Listing 4.12 shows a bug in how the processor saves the PC for Illegal Instruction Exceptions, due to the delay slot problem.

Software that has a delay slot instruction interrupted by an exception that activates bug13 will not return from the exception handler to the correct address. Most of the time, the branch/jump instruction is effectively ignored as the exception returns to the delay slot. This corrupts the control flow of software.

We do not recover from this bug because the bug breaks the exception support logic that Erratacator relies on to bridge detection and recovery. In general, recovery is possible. In the general case, the recovery firmware simulates taking the exception, updating software's virtualized state accordingly. The recovery firmware then returns control to the software stack, which is now inside an exception handler, but with the correct value in EPCR.

bug16

This bug causes the processor to accept data from the bus even though the cache is enabled and there is a cache hit. Listing 4.13 shows both the buggy lines and the correct lines for both the data and instruction cache. Because both the instruction and the data cache are affected by the bug, software will see both corrupted instructions and corrupted data loads. Software triggers this bug when it enables the cache and subsequently loads a data or instruction word that is resident in the cache. This is a very timing critical bug in that there must be a pending request on the bus that is serviced just as software enables the cache, which reports a hit.

Since this is such a timing-sensitive bug, taking the trap created by the detector and returning control to software is enough to perturb the timing of events to avoid the bug.

In or1200_ic_fsm.v

```
// Asserted when a cache hit occurs and the first word is ready/valid
assign first_hit_ack = (state == `OR1200_ICFSM_CFETCH) & hitmiss_eval &
                       !tagcomp_miss & !cache_inhibit;
// BUG: Bus data takes precedence over cache when cache enabled
// assign first_miss_ack = (state == `OR1200_ICFSM_CFETCH) & biudata_valid;
// FIX: Cache overpowers bus when it is enabled
// Asserted when a cache miss occurs, but the first word of the new
// cache line is ready (on the bus)
// Cache hits overpower bus data
assign first_miss_ack = (state == `OR1200_ICFSM_CFETCH) & biudata_valid &
                        ~first_hit_ack;
```

In or1200_dc_fsm.v

```
// BUG: Bus has precedence over enabled cache
// assign first_miss_ack = load_miss_ack | load_inhibit_ack;
// first_hit_ack takes precedence over first_miss_ack
assign first_miss_ack = ~first_hit_ack & (load_miss_ack | load_inhibit_ack);
```

Listing 4.13: *bug16 source code*

4.6 Discussion

4.6.1 Extendability

The detection technique is generally applicable to all sequential hardware. The paper that proposed the processor bug detection mechanism implemented by Erratacator explains and provides examples of how it is possible to always link the value of any wire/signal in arbitrary hardware to a set of register (i.e., flip-flop) values. As shown in Erratacator's evaluation, this may lead to false positives.

Recovery is more complex as there needs to be a way for software/firmware to emulate the behavior of the buggy hardware. Recovering from bug detections internal to a processor rely on redundancy within the ISA. Recovering from bus and memory bug detections relies on the ability to use different-sized memory access instructions. Recovering from bug detections in performance-only related components of the processor, e.g., cache and TLB, can default to disabling the component until it is safe to enable it. Recovering from bugs in peripherals (e.g., Ethernet controller) relies on the interface that the periph-

eral exposes to software. The Erratacator evaluation contains bugs internal to the processor and bugs in performance-only components. Erratacator also supports recoding IO/memory accesses.

4.6.2 Using other processor bug detectors

One of the contributions of Erratacator is creating a complete system with a single-exception-wire detector interface. Processor designers can connect the output of whatever processor bug detection mechanism that best meets their needs to this interface. As long as the bug detector provides low latency detection, it works.

We do not go into detail on alternative bug detection techniques because it is not our focus.

4.6.3 Erratacator's effects on system security

Adding components to a system increases functionality at the cost of increased complexity, which can negatively impact the security of the system. Thus, when adding a component to a system, it is critical to re-evaluate the security of that system. In this section, we examine potential security vulnerabilities created by adding our bug detectors and recovery firmware to a system. We break the discussion into two cases: security vulnerabilities resulting from the new components, assuming correct configuration, and vulnerabilities arising from misconfigurations of the system.

In analyzing Erratacator for new security vulnerabilities, we identify two high level concerns: 1) can the attacker observe, from the ISA level of abstraction, sub-ISA-level state of the system and 2) can the attacker gain any new control over the system. In terms of opening up new vectors of system control, we observe that in an Erratacator protected system (assuming a correctly configured system) attackers generally have no more control than in an unprotected system. At most, an attacker can issue instruction sequences that activate Erratacator's bug detectors, which then invokes recovery. In the worst case, this effectively locks the system for the attacking process's time slice—a self denial-of-service.

With respect to revealing sub-ISA-level state of the system, Erratacator

increases the amount of information available to attackers. Recall that Erratacator's goal is to reinforce the ISA: supporting software's assumption of a perfect processor. The problem is that the instruction set specification does not completely specify the run-time state of the system, even at the ISA level. This means that there exists, at run time, low level processor state visible to software that Erratacator does not simulate. For instance, the ISA provides no notion of time and thus, different implementations of the specification are free to differ in how long it takes a given instruction to execute. This also means that an attacker can use differences in the time taken by different instruction sequences to find out if they are running on an Erratacator protected system and what protections are in place. The current implementation of Erratacator does not attempt to hide from the software stack (but it is protected from the software stack), so it is trivial for an attacker to detect its presence.

We could attempt to hide from the software stack by augmenting the design of Erratacator so that it virtualizes the timer register, but curious users could still use a remote time server and cleverly crafted instruction sequences to infer whether the system was protected by Erratacator or not. Note that this weakness is not distinct to Erratacator, but a general problem faced when employing any form of hardware virtualization [92, 93]. So, instead of adding the overhead and complexity, we assume that the software stack knows what protections are in place. Even when the attacker knows what protections are in place, they have *no* more information and gain *no* additional control compared to an unprotected system because attackers already have access to errata documents.

We now consider the security implications of a misconfigured system. A misconfiguration of the detectors via a corrupted system signature or system mask can result in missed bugs activations and an increased chance of false detections. Missing bug activations leaves the system vulnerable to attack if the user can identify the misconfiguration and then exploit the processor bug. An increased risk of false detections results in unneeded invocations of the recovery firmware, adding software run time overhead. Taken to the extreme, a corrupted system signature or system mask does not make an Erratacator protected system more vulnerable than an unprotected system, just slower.

Another possible misconfiguration is of the recovery firmware. While it is

safe to assume that the recoding routines are correct—since we formally verify them—the entry, instruction processing (*i.e.,* fetch from software stack and decode), and exit routines are vulnerable to misconfiguration. Errors in these routines appear as processor bugs to the software stack and can manifest in the same ways as processor bugs. Given that coding these routines is the least automated and most complex part of building the recovery firmware, these routines are a prime place for bugs. In fact, we experienced several bugs in the entry and exit routines while building Erratacator. The critical, but buggy nature of the entry and exit routines and their regular structure motivates the use of formal verification to avoid misconfiguration.

The worst case misconfiguration is allowing unrestricted access to Erratacator's components. In such cases, attackers can use Erratacator's bug detectors to gain low-level insight on the state of the processor or to efficiently interpose on other processes and the operating system. Since Erratacator is essentially a lightweight hypervisor, an attacker controlling Erratacator is equivalent to the Virtual Machine-based Rootkit proposed by King and colleagues [92]. We protect Erratacator by separating it from the software stack using hardware fences that prevent memory contamination and privilege checks on accesses to the detectors and other Erratacator support registers. These protections assume that software is correct, thus any security vulnerabilities in software are an opportunity for an attacker to compromise Erratacator.

4.7 Conclusion

The key to Erratacator's practicality is leveraging the insight that processors have sufficient innate functionality to detect and recover from processor bugs. Low latency detection enables the processor pipeline to flush contaminated state before it makes it to the software level, fulfilling the role of a checkpointing and rollback mechanism. The redundancy of processor functionality enables the replacement of multiple redundant executions or multiple redundant executors. Finally, allowing software to manage its own reliability exposes a trade-off between run time overhead and contamination risk.

Chapter 5

BlueChip

Modern hardware design processes closely resemble the software design process. Hardware designs consist of millions of lines of code and often leverage libraries, toolkits, and components from multiple vendors. These designs are then "compiled" (synthesized) for fabrication. As with software, the growing complexity of hardware designs creates opportunities for hardware to become a vehicle for malice. Recent work has demonstrated that small malicious modifications to a hardware-level design can compromise the security of the entire computing system [4].

Malicious hardware has two key properties that make it even more damaging than malicious software. First, hardware presents a more persistent attack vector. Whereas software vulnerabilities can be fixed via software update patches or reimaging, fixing well-crafted hardware-level vulnerabilities would likely require physically replacing the compromised hardware components. A hardware recall similar to Intel's Pentium FDIV bug (which cost 500 million dollars to recall five million chips) has been estimated to cost many billions of dollars today [94]. Furthermore, the skill required to replace hardware and the rise of deeply embedded systems ensure that vulnerable systems will remain in active use after the discovery of the vulnerability. Second, hardware is the lowest layer in the computer system, providing malicious hardware with control over the software running above. This low-level control enables sophisticated and stealthy attacks aimed at evading software-based defenses.

Such an attack might use a special, or unlikely, event to trigger deeply buried malicious logic which was inserted during design time. For example, attackers might introduce a sequence of bytes into the hardware that activates the malicious logic. This logic might escalate privileges, turn off access control checks, or execute arbitrary instructions, providing a path for the malefactor to take control of the machine. The malicious hardware thus

provides a *foothold* for subsequent system-level attacks.

In this dissertation we present the design, implementation, and evaluation of BlueChip, a hybrid design-time/runtime system for detecting and neutralizing malicious circuits. During the design phase, BlueChip flags as suspicious, any unused circuitry (any circuit not activated by any of the many design verification tests) and deactivates them. However, these seemingly suspicious circuits might actually be part of a legitimate circuit within the design, so BlueChip inserts circuitry to raise an exception whenever one of these suspicious circuits would have been activated. The exception handler software is responsible for emulating hardware instructions to allow the system to continue execution. BlueChip's overall goal is to push the complexity of coping with malicious hardware up to a higher, more flexible, and adaptable layer in the system stack.

The contributions of this reference implementation are:

- We present the BlueChip system (Sections 5.2 and 5.3), which automatically removes potentially malicious circuits from a hardware design and uses low-level software to emulate around removed hardware.
- We propose an algorithm (Section 5.4), called *unused circuit identification*, for automatically identifying circuits that avoid affecting outputs during design verification. We demonstrate its feasibility (Section 5.5) for use in addressing the problem of detecting malicious hardware.
- We demonstrate (Sections 5.6, 5.7, and 5.8), using fully-tested malicious hardware modifications as test cases on a SPARC processor implementation operating on an FPGA, that: (1) the system successfully prevents three different malicious hardware modifications, and (2) the performance effects (and hence the overhead) of the system are small.

5.1 Motivation and attack model

This dissertation focuses on the problem of malicious circuits introduced during the hardware design process. Today's complicated hardware designs are increasingly vulnerable to the undetected insertion of malicious circuitry to create a hardware trojan horse. In other domains, examples of this general type of intentional insertion of malicious functionality include compromises of

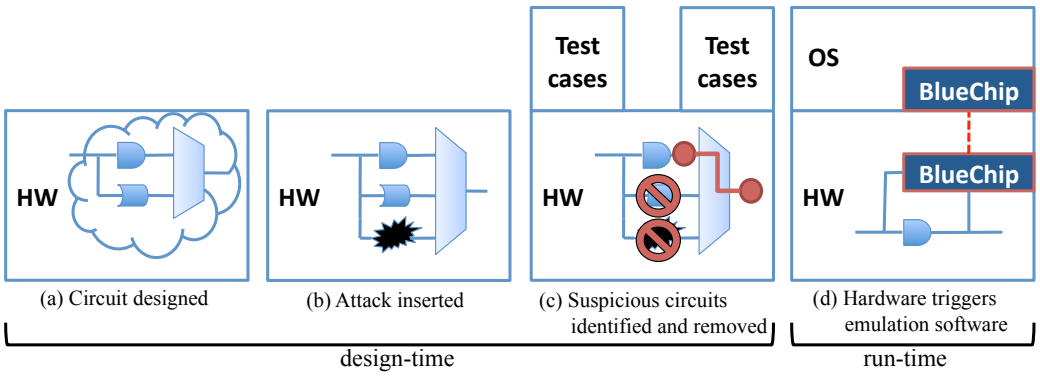


Figure 5.1: *Overall BlueChip architecture. This figure shows the overall flow for BlueChip where (a) designers develop hardware designs and (b) a rogue designer inserts malicious logic into the design. During design verification phase, (c) BlueChip identifies and removes suspicious circuits and inserts runtime hardware checks. (d) During runtime, these hardware checks invoke software exceptions to provide the BlueChip software an opportunity to advance the computation by emulating instructions, even though BlueChip may have removed legitimate circuits.*

---

software development tools [95], system designers inserting malicious source code intentionally [96, 97, 98], compromised servers that host modified source code [99, 100], and products that come pre-installed with malware [101, 102, 103]. Such attacks introduce little risk of punishment, because the complexity of modern systems and prevalence of unintentional bugs makes it difficult to prove malice or to correctly attribute the problem to its source [104].

More specifically, our threat model is that a rogue designer covertly adds trojan circuits to a hardware design. We focus on two possible scenarios for such rogue insertion. First, one or more disgruntled employees at a hardware design company surreptitiously and intentionally inserts malicious circuits into a design prior to final design validation with the hope that the changes will evade detection. The malicious hardware demonstrated by King *et al.* [4] support the plausibility of this scenario, in that only small and localized changes (*e.g.*,, to a single hardware source file) are sufficient for creating powerful malicious circuits designed for bootstrapping larger system-level attacks. We call such malicious circuits *footholds*, and such footholds persist even after malicious software has been discovered and removed, giving attackers a permanent vector into a compromised system.

The second scenario is enabled by the trend toward "softcores" and other pre-designed hardware IP (intellectual property) blocks. Many system-on-

chip (SoC) designs aggregate subcomponents from existing commercial or open-source IP. Although generally trusted, these third-party IP blocks may not be trustworthy. In this scenario, an attacker can create new IP or modify existing IP blocks to add malicious circuits. The attacker then distributes or licenses the IP in the hope that some SoC creator will incorporate it and include it in a fabricated chip. Although the SoC creator will likely perform significant design verification focused on finding design bugs, traditional black-box design verification is unlikely to reveal malicious hardware.

In either scenario, the attacker's motivation could be financial or general malice. If the design modification remains undetected by final design validation and verification, the malicious circuitry will be present in the manufactured hardware that is shipped to customers and integrated into computing systems. The attacker has achieved this without the resources necessary to actually fabricate a chip or otherwise attacking the manufacturing and distribution supply chain. We assume that only one or a few individuals are acting maliciously (*i.e.*,, not the entire design team) and that these individuals are unable to compromise the final end-to-end design verification and validation process, which is typically performed by a distinct group of engineers.

Our approach to detecting insertions of malicious hardware assumes analysis at the level of a hardware netlist or hardware description language (HDL) source. In the two scenarios outlined, this assumption is reasonable, as (1) design validation and verification is primarily performed at this level and (2) softcore IP blocks are often distributed in HDL or netlist form.

We assume the system software is trustworthy and non-malicious (although the malicious hardware may attempt to subvert the overlying software layers).

5.2 The BlueChip approach

Our overall BlueChip architecture is shown in Figure 5.1. In the first phase of operation, BlueChip analyzes the circuit's behavior during design verification to identify candidate circuits that might be malicious. Once BlueChip identifies a suspect circuit, BlueChip automatically removes the circuit from the design. Because BlueChip might remove legitimate circuits as part of the transformation, it inserts logic to detect if the removed circuits would have been activated, and triggers an exception if the hardware encounters this con-

dition during runtime. The hardware delivers this exception to the BlueChip software layer. The exception handling software is responsible for recovering from the fault and advancing the computation by emulating the instruction that was executing when the exception occurred. BlueChip pushes much of the complexity up to the software layer, allowing defenders to rapidly refine defenses, turning the permanence of the hardware attack into a disadvantage for attackers.

BlueChip can operate in spite of removed hardware because the removed circuits operate at a lower layer of abstraction than the software emulation layer responsible for recovery. BlueChip software does not emulate the removed hardware directly. Instead, BlueChip software emulates the effects of removed hardware using a simple, high-level, and implementation-independent specification of hardware, *i.e.*,, the processor's instruction-set-architecture specification. The BlueChip software emulates the effects of the removed hardware by emulating one or more instructions, updating the processor registers and memory values, and resuming execution. The computation can generally make forward progress despite the removed hardware logic, although software emulation of instructions is slower than normal hardware execution.

In some respects our overall BlueChip system resembles floating point instruction emulation for processors that omit floating point hardware. If a processor design omits floating point unit (FPU) hardware, floating point instructions raise an exception that the OS handles. The OS can emulate the effects of the missing hardware using available integer instructions. Like FPU emulation, BlueChip uses software to emulate the effects of missing hardware using the available hardware resources. However, the hardware BlueChip removes is *not* necessarily associated with specific instructions and can trigger BlueChip exceptions at unpredictable states and events, presenting a number of unique challenges that we address in Section 5.3.

5.3 BlueChip design

This section describes the design of BlueChip. We discuss the BlueChip hardware component (Section 5.3.1), the BlueChip software component (Section 5.3.2), and possible alternative architectures (Section 5.3.3). Section 5.4

discusses our algorithm for identifying suspicious circuits and Section 5.5 describes how BlueChip uses these detection results to modify the hardware design.

We present general requirements for applying BlueChip to hardware and to software, but we describe our specific design for a modified processor and recovery software running within an operating system.

5.3.1 BlueChip hardware

To apply BlueChip techniques, a hardware component must be able to meet three general requirements. First, BlueChip requires a hardware exception mechanism for passing control to the software. Second, BlueChip must prevent modified hardware state from committing when the hardware triggers a BlueChip exception. Third, to enable recovery the hardware must provide software access to hardware state, such as processor register values and other architecturally visible state.

Processors are well suited to meet the requirements for BlueChip because they already have many of the mechanisms BlueChip requires. Processors provide easy access to architecturally visible states to enable context switching, and processors have existing exception delivery mechanisms that provide a convenient way to pass control to a software exception handler. Also, most processors support precise exceptions that include a lightweight recovery mechanism to prevent committing any state associated with the exception. As a result, we use existing exception handling mechanisms within the processor to deliver BlueChip exceptions.

One modification we make to current exception semantics is to assert BlueChip exceptions immediately instead of associating them with individual instructions. In current processors, many exceptions are associated with a specific instruction that caused the fault. However, in BlueChip it is often unclear which individual instruction would have triggered the removed logic. Thus, BlueChip asserts the exception immediately, flushes the pipeline, and passes control to the BlueChip software.

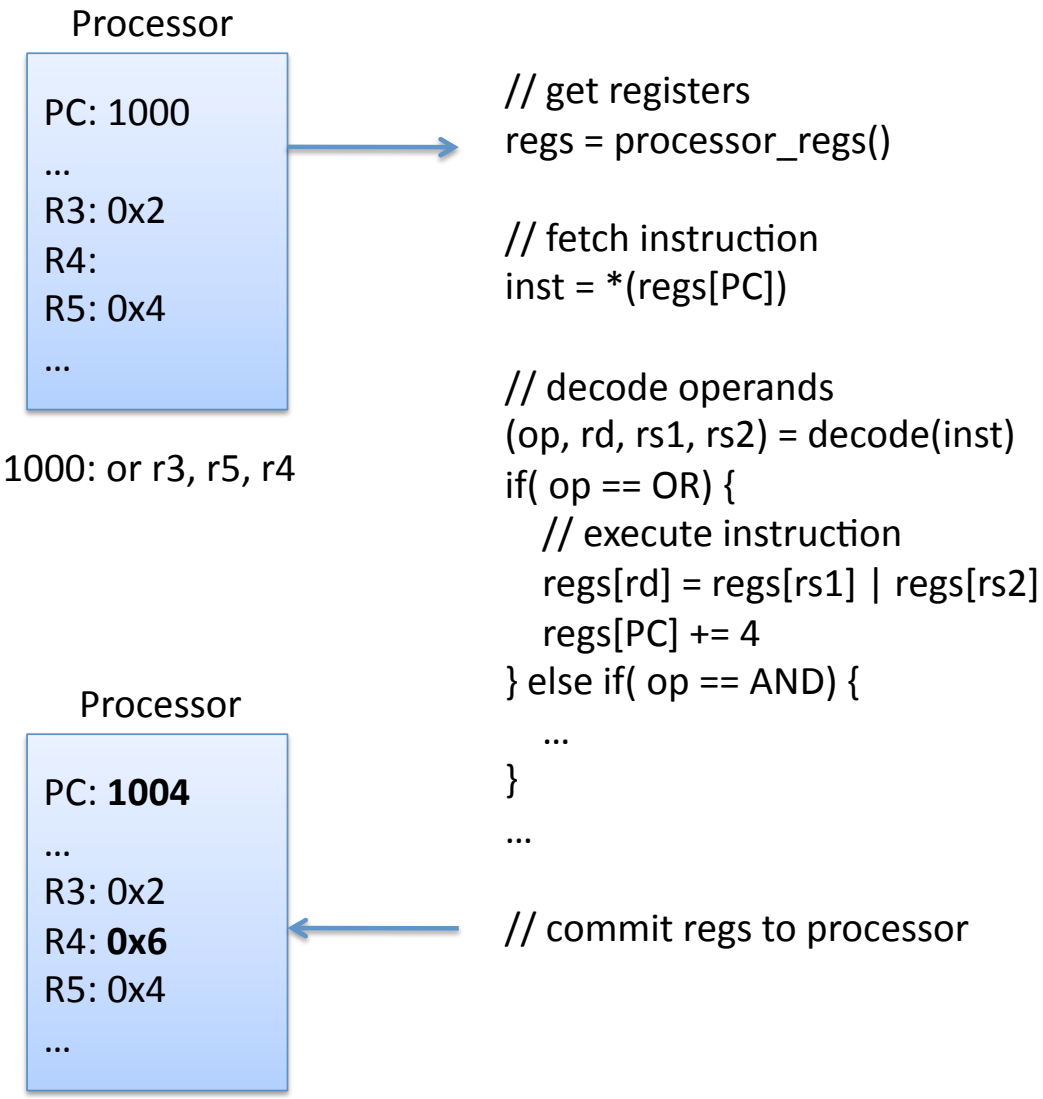


Figure 5.2: *Basic flow for an instruction-by-instruction emulator. This figure shows how a software emulator can calculate the changes to processor registers induced by an* `or` *instruction.*

5.3.2 BlueChip software

The BlueChip software is responsible for recovering from BlueChip hardware exceptions and providing a mechanism for system forward progress. This responsibility presents unusual design challenges for the BlueChip software because it runs on a processor that has had some portions of the design removed, therefore some features of the processor may be unavailable to the BlueChip exception handler software.

To handle BlueChip exceptions, BlueChip uses a recovery technique where

the BlueChip software emulates faulting instructions to carry out the computation. The basic emulation strategy is similar to an instruction-by-instruction emulator, where for each instruction, BlueChip software reads the instruction from memory, decodes the instruction, calculates the effects of the instruction, and commits the register and memory changes (Figure 5.2). By emulating instructions in software, BlueChip skips past the instructions that use the removed hardware, duplicating their effects in software, providing the opportunity for the system to continue making forward progress despite the missing circuits.

One problem with our first implementation of this basic strategy was that our emulation routines sometimes depended on removed hardware, thus causing unrecoverable recursive BlueChip exceptions. For example, the "shadow-mode attack" (Section 5.6) uses a "bootstrap trigger" that initiates the attack. The bootstrap trigger circuit monitors the values going into the data cache and enables the attack once it observes a specific value being stored in memory. This specific value will always trigger the attack regardless of the previous states and events in the system. After BlueChip identified and removed the attack circuits, the BlueChip hardware triggered an exception whenever the software attempted to store the attack value to a memory location. Our first implementation of the `store` emulation code simply re-issued a `store` instruction with the same address and value to emulate the effects of the removed logic, thus creating an unrecoverable recursive exception.

To avoid unrecoverable recursive BlueChip exceptions, we emulate around faulting instructions by producing semantically equivalent results while avoiding BlueChip exception states. For ALU operations, we map the emulated instructions to an alternative set of ALU operations and equivalent, but different, operand values. For example, we implement `or` emulation using a series of `xor`, and `nand` instructions rather than executing an `or` to perform OR operations (Figure 5.3). For `load` and `store` instructions we have somewhat less flexibility because these instructions are the sole means for performing I/O operations that access off-processor memory. However, we do perform some transformations, such as emulating word sized accesses using byte sized accesses and vice versa.

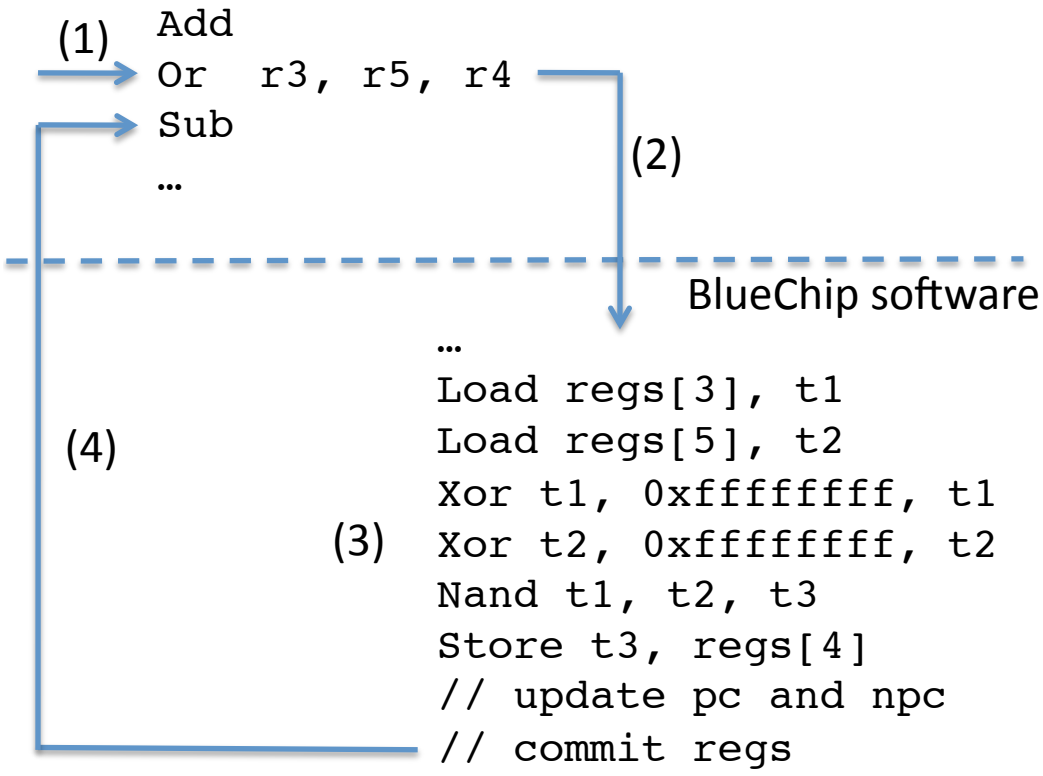


Figure 5.3: *BlueChip emulation. This figure shows how BlueChip emulates around removed hardware. First, (1) the software executes an* `or` *instruction, (2) which causes a BlueChip exception. This exception is handled by the BlueChip software, (3) which emulates the* `or` *instruction using* `xor` *and* `nand` *instructions (4) before returning control to the next instruction in the program.*

5.3.3 Alternative designs

In this section we discuss other possible designs and some of the trade offs inherent in their design decisions. BlueChip delivers exceptions using existing processor exception handling mechanisms. One alternative could be adding new hardware to deliver exceptions to the BlueChip software component. In our current design, BlueChip is constrained by the semantics of the existing exception handling mechanisms and cannot deliver exceptions when software disables interrupts.

An alternative approach could have been to add extra registers and logic to the processor to allow BlueChip to save state and recover from BlueChip exceptions, even when the software disables interrupts. However, this additional state and logic would have required encoding several implementation-specific details of the hardware design into BlueChip, potentially making it

more difficult to insert BlueChip logic automatically. Given the infrequency of disabling interrupts for long periods in modern commodity operating systems, we decided to use existing processor exception delivery mechanisms. If a particular choice of hardware and software makes this unacceptable, there are several straightforward approaches to addressing this issue, such as using a hypervisor with some additional support for BlueChip.

BlueChip emulates around BlueChip exceptions by using different instructions to emulate computations that depend on hardware removed by BlueChip. In our current design we implement this emulation technique manually for all instructions in the processor's instruction set. However, we still rely on portions of the OS and exception handling code to save and restore the system states we emulate around. It might be possible for these instructions to inadvertently invoke an unrecoverable BlueChip exception by executing an instruction that causes a BlueChip exception. One way to avoid unrecoverable BlueChip exceptions could be to modify the compiler to emit only a small set of Turing complete instructions for the BlueChip software, such as `nand`, `load`, and `store` instructions. Then we could focus our testing or formal methods efforts on this subset of the instruction set to decrease the probability of an unrecoverable BlueChip exception. This technique would likely make the BlueChip software slower because it potentially uses more instructions to carry out equivalent computations, but it could decrease the occurrence of unrecoverable BlueChip exceptions.

5.4 Detecting suspicious circuits

This section describes our detection algorithm for identifying suspicious circuits automatically within a hardware design. We focus on automatically detecting potentially malicious logic embedded within the HDL source code of a design, and we perform our detection during the design phase of the hardware design life cycle.

Our goal is to develop an algorithm that identifies malicious circuits without identifying benign circuits. In addition, our technique should be difficult for an attacker to avoid, and it should identify potentially malicious code automatically without requiring the defender to develop a new set of design verification tests specifically for our new detection algorithm.

Hardware designs often include extensive design verification tests that designers use to verify the functionality of a component. In general, test cases use a set of inputs and verify that the hardware circuit outputs the expected results. For example, test cases for processors use a sequence of instructions as the input, with the processor registers and system memory as outputs.

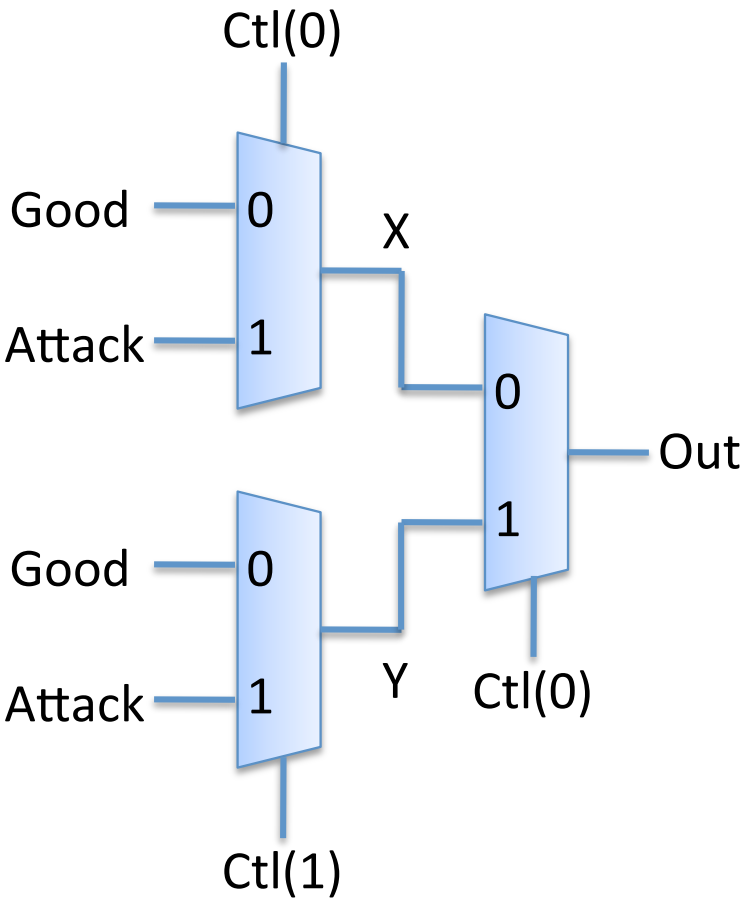
Our approach is to use design verification tests to help detect attack circuits. If an attack circuit contaminates the output for a test case, the designer would know that the circuit is operating out-of-spec, potentially detecting the attack. However, recent research has shown how hardware attacks can be implemented using small circuits that are designed not to trigger during routine testing [4]. This evasion technique works by guarding the attack circuit with triggering logic that enables the attack only when it observes a specific sequence of events or a specific data value (*e.g.,*, the attack triggers only when the hardware encounters a predefined 128-bit value). This attack-hiding technique works because malicious hardware designers can avoid perturbing outputs during testing by hiding deep within the vast state space of a design,[1] but can still enable attacks in the field by inducing the trigger sequence. Our proposal is to consider circuits suspicious whenever they are included in a design but do *not* affect any outputs during testing.

5.4.1 Straw-man approach: code coverage

One possible approach to identifying potentially malicious circuits could be to use code coverage. Code coverage is defined as the percentage of lines of code that are executed, out of those possible. Because attackers will likely try to avoid affecting outputs during testing, highlighting uncovered lines of code seems like a viable approach to identifying potentially malicious circuits.

An attacker can easily craft circuits that are covered completely by testing, but never trigger an attack. For example, Figure 5.4 shows a multiplexer (mux) circuit that can be covered fully without outputting the attack value. If the verification test suite includes control states 00, 01, and 10, all lines of code that make up the circuit will be covered, but the output will always be "Good". We apply this evasion technique for some of the attacks we evaluated (Section 5.6) and find that it does evade code coverage detection.

[1] A processor with 16 32-bit registers, a 16k instruction cache, a 64k data cache, and 300 pins has *at least* 2^{655872} states, and up to 2^{300} transition edges.




```
X <= (Ctl(0) = '0') ? Good : Attack
Y <= (Ctl(1) = '0') ? Good : Attack
Out <= (Ctl(0) = '0') ? X : Y
```

Figure 5.4: *Circuit diagram and HDL source code for a mux that can pass code coverage testing without enabling the attack. This figure shows how a well-crafted mux can pass coverage tests when the appropriate control states (Ctl(0) and Ctl(1)) are triggered during testing. Control states 00, 01, and 10 will fully cover the circuit without triggering the attack condition.*

Although code coverage can complicate the attacker's task of avoiding testing, this technique can be defeated because code coverage misses the fundamental property of malicious circuits: attackers are likely to avoid affecting outputs during testing, otherwise they would be caught. Instead, what defenders need is a technique that zeros in on this property to identify potentially malicious circuits more reliably.

```
// step one: generate data-flow graph
// and find connected pairs
pairs = {connected data-flow pairs}

// step two: simulate and try to find
// any logic that does not affect the
// data-flow pairs
foreach simulation clock cycle
  foreach pair in pairs
    if the sink and source not equal
      remove the pair from the pairs set
```

Figure 5.5: *Identifying potentially malicious circuits using our UCI algorithm.*

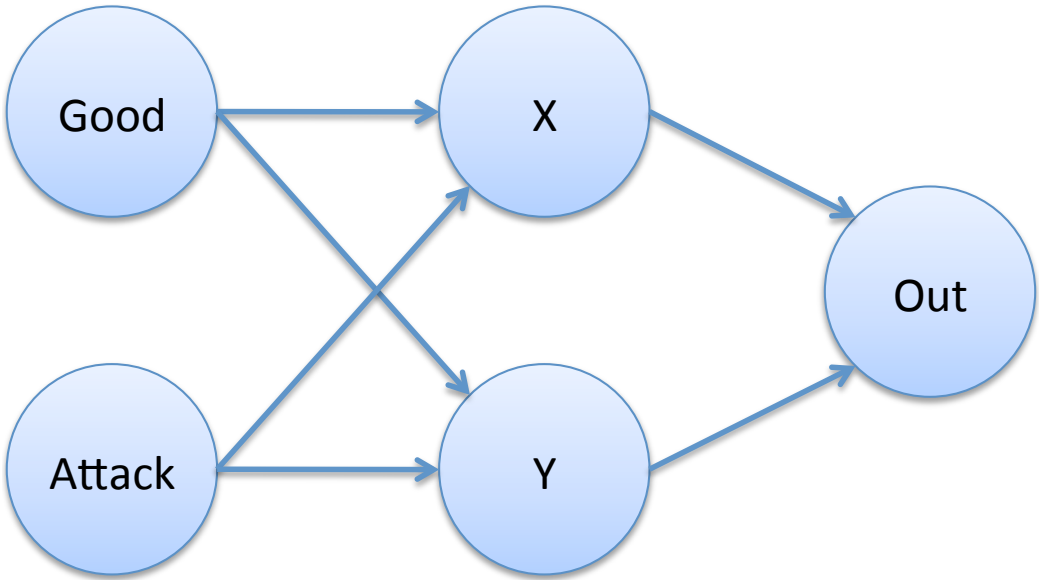


Figure 5.6: *Data-flow graph for mux replacement circuit.*

5.4.2 Unused circuit identification

This section describes our algorithm, called unused circuit identification (UCI), for identifying potentially malicious circuits at design time. Our technique focuses on identifying portions of the circuit that do not affect outputs during testing.

To identify potentially malicious circuits, our algorithm performs two steps (Figure 5.5). First, UCI creates a data-flow graph for our circuit (Figure 5.6). In this graph, nodes are signals (wires) and state elements; edges indicate data flow between the nodes. Based on this data-flow graph, UCI generates a

list of all signal pairs, or *data-flow pairs*, where data flows from a source signal to a sink signal. This list of data-flow pairs includes both direct dependencies (*e.g.,*, (Good, X) in Figure 5.6) and indirect dependencies (*e.g.,*, (Good, Out) in Figure 5.6).

Second, UCI simulates the HDL code using design verification tests to find the set of data-flow pairs where intermediate logic does not affect the data that flows between the source and sink signals. To test for this condition, at each simulation step UCI checks for inequality for each of our remaining data-flow pairs. If the elements of a pair are not equal, this implies, conservatively, that the logic in between the two pairs has an effect on the value, thus we remove pairs with unequal elements from our data-flow-pairs set. For registers, UCI accounts for latched data by maintaining a history of simulation values, allowing it make the appropriate comparison for tracking signal propagation.

After the simulation completes, UCI has a set of remaining data-flow pairs where the logic in between the pairs does not affect the signal value from source to sink. In other words, we could replace the intermediate logic with a wire, possibly including some delay registers, and it would not affect the overall behavior of the circuit in any way for our design verification tests.

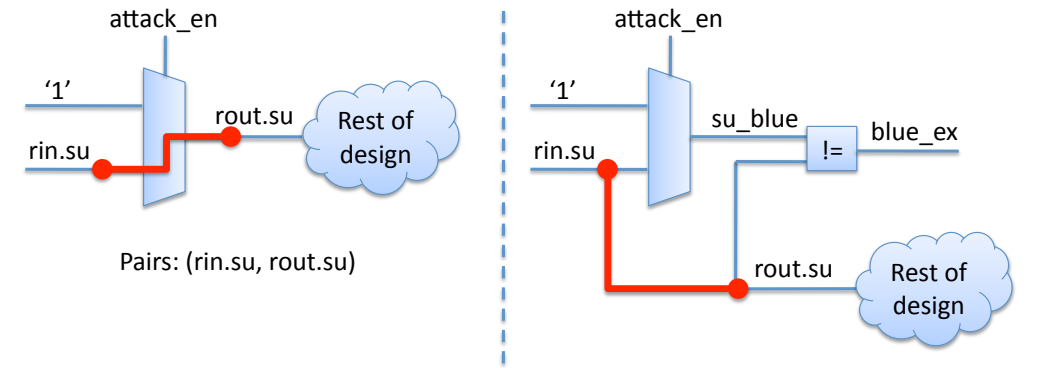
Consider how this algorithm works for the mux-replacement circuit shown in Figure 5.4:

1. UCI initially creates the set of all possible data-flow pairs, which for our circuit is (Good,X), (Attack,X), (Good,Y), (Attack,Y), (Good,Out), (Attack,Out), (X,Out), and (Y,Out).

2. UCI considers the first simulation step where the control signals are 00 and the output is Good, X is Good, and Y is Good. This removes (Attack,X), (Attack,Y), and (Attack,Out).

3. UCI considers the second simulation step where the control signals are 01 and the output is Good, X is Good, and Y is Attack. This removes (Good,Y) and (Y,Out).

4. UCI considers the third simulation step where the control signals are 10 and the output is Good, X is Attack, and Y is Good. This removes (Good,X) and (X, Out).



(a) UCI identifies attack

(b) BlueChip transforms HDL to remove attack

Figure 5.7: *HDL code and circuit diagram for HDL transformations BlueChip makes to remove an attack from a design. This figure shows (a) the original design where an attacker can transition the processor into supervisor mode by asserting the* `attack_en` *signal. During design verification, UCI detects that the value for* `rout.su` *always equals* `rin.su`*, thus identifying the mux as a candidate for removal. Part (b) shows how BlueChip removes this logic by connecting* `rin.su` *to* `rout.su` *directly and adds exception notification logic to notify software of any inconsistencies at runtime.*

---

5. UCI finishes the simulation and is left with (Good,Out) in the list of data-flow pairs where intermediate logic does not affect the signal propagation.

The resulting output from UCI for this example identifies the malicious circuit without identifying any additional signals. Because it systematically identifies circuits that avoid affecting outputs during testing, BlueChip connects the "Good" signal directly to the "Out" signal, thus removing the malicious elements from the design.

5.4.3 UCI limitations

In Section 5.8 we show that UCI successfully identifies the malicious circuits for the hardware-level footholds we developed. In this section we discuss ways an attacker could hide malicious circuits from UCI.

First, an attacker could include malicious test cases that check attack states incorrectly. By including these malicious test cases in the design verification

test suite an attacker could fool the system designer into thinking that the out-of-spec modifications are in fact in-spec, thus slipping the attack past UCI. However, design verification tests work at a higher level of abstraction, making it easier for system designers to verify the results independently. In fact, the BlueChip software includes code for instruction-level emulation of the processor's instruction set, and we use this emulation code on our test cases to verify that the test suite checks states correctly.

Second, an attacker could avoid UCI by crafting malicious circuits that affect unchecked outputs. Unchecked outputs could arise from incomplete test cases or from unspecified output states. For example, the memory model in the SPARC processor specification provides freedom for an implementation to relax consistency between processors in a multi-processor system [105]. This type of implementation-specific behavior could be used by an attacker who affects outputs that might be difficult for a testing program to check deterministically, thus causing malicious circuits to affect outputs and avoid our analysis. However, this evasion technique requires the attacker to trigger the attack during design verification, thus tests that check implementation-specific states and events could detect the foothold directly.

Third, to simplify the analysis of the HDL source, our current UCI implementation excludes mux control signals from the data-flow graph. If an attacker could use only mux control signals to modify architecturally visible states directly, UCI would miss the attack. However, this limitation is only an artifact of our current implementation and would likely be remedied if we included mux control signals in our analysis.

5.5 Using UCI results in BlueChip

This section discusses how BlueChip uses the results from UCI to eliminate the effects of suspicious circuits. UCI outputs a set of data-flow pairs, where each pair has a source element, a sink element, and a delay element. Conceptually, the source and the sink element can be connected directly by a wire containing delay number of registers, effectively short circuiting the signals with a delay line. This removes the effects of any intermediate logic between the source and the sink. BlueChip implements this short-circuit by performing a source-to-source transformation on the design's HDL source

code.

Once UCI generates a set of data-flow pairs, the pairs are fed into the BlueChip system (transformation pictured in Figure 5.7). BlueChip takes these pairs as an input and replaces the suspicious circuits by modifying the HDL code using three steps. First, BlueChip creates a backup version of each sink in the list of pairs. The backup version of a signal holds the value that would have been assigned to the signal in the original circuit. Second, BlueChip adds a new assignment to the original signal. The value assigned to the original signal is simply the value of the source element in the pair, creating a short circuit between the source and the sink. The rest of the design will see this short-circuited value, while the BlueChip exception generation hardware sees the backed-up version. The third and final step consists of adding the BlueChip exception generation logic to the source file. This circuit compares the backed-up value of the sink element with the source value. BlueChip generates an exception whenever any of these comparisons are not true. When a BlueChip exception occurs, it signals that the hardware was about to enter a state that was not seen during testing. From here, the BlueChip software is responsible for making forward progress.

The HDL transformation algorithm also must handle multiple data-flow pairs that share the same sink signal but have different sources. This situation could potentially cause problems for BlueChip because it is unclear which source signal to short-circuit to the original sink signal. Our solution is to pick one pair for the sink signal assignment, but include exception generation logic for all pairs that share the same sink element. This means that all violations are detected, but the sink may be shorted with the source that caused the exception. This untested state is safe because (1) BlueChip asserts exceptions immediately when detecting an inconsistency between the original design and the modified circuit, (2) BlueChip checks all data-flow pairs for inconsistencies, and (3) BlueChip leverages hardware recovery mechanisms to prevent the persistence of untrusted state modifications.

5.6 Malicious hardware footholds

This section describes the malicious hardware trojans we used to test the effectiveness of BlueChip. Prior work on developing hardware attacks focused

on adding minimal additional logic gates as a starting point for a system-level attack [4]. We call this type of hardware mechanism a *foothold*. We explored three such footholds. The first foothold, called the *supervisor transition* foothold, enables an attacker to transition the processor into supervisor mode to escalate the privileges of user-mode code. The second foothold, called the *memory redirection* foothold, enables an attacker to read and write arbitrary virtual memory locations. The third foothold, called the *shadow mode* foothold, enables an attacker to pass control to invisible firmware located within the processor and take control of the system. Previous work has shown how these types of footholds can be used as part of a system-level attack to carry out high-level, high-value attacks, such as escalating privileges of a process or enabling attackers to login to a remote system automatically [4].

5.6.1 Supervisor transition foothold

Our supervisor transition foothold provides a hardware mechanism that allows unprivileged programs to transition the processor into supervisor mode. This transition grants access to privileged instructions and bypasses the usual hardware-enforced protections, allowing the attacker to escalate the privileges of an otherwise unprivileged process.

The malicious hardware is triggered when it observes a sequence of instructions being executed by the processor. This attack sequence can be arbitrarily long to avoid false positives, and the particular sequence is hard coded into the hardware. This foothold requires relatively few transistors. We implement it by including a small state machine in the integer unit (*i.e.*,, the pipeline) of the processor that looks for the triggering sequence of instructions, and asserts the supervisor-mode bit when enabled.

5.6.2 Memory redirection foothold

Our memory redirection foothold provides hardware support for unprivileged malicious software by allowing an attacker to access arbitrary virtual memory locations.

This foothold uses a sequence of bytes as the trigger. In this case, when

the foothold observes store instructions with a particular sequence of byte values it then interprets the subsequent bytes as the *redirection address*. The malicious logic records the address of the block and the redirection address in hardware registers. The next time the address is loaded from memory, the malicious hardware substitutes the redirection address as the address to be loaded and asserts the supervisor bit passed to the memory management unit (MMU). That is, the next read to this block will return the value of a different location in the memory. Memory writes are handled analogously, in that the next write to the block is redirected to write to the redirection address. The net effect is providing full access to arbitrary virtual memory locations and bypassing MMU protections enforced in the processor.

This foothold provides flexibility for attackers because attackers can trigger the circuit using only data values. Attackers can trigger the foothold by injecting specific data values into a system using a number of techniques including unsolicited network packets, emails, and images on web sites. By using these mechanisms to arbitrarily manipulate the system's memory, a remote attacker can compromise the system, for example, by searching memory for encryption keys, disabling authentication checks by modifying the memory of the targeted system, or altering executable code running on the system.

5.6.3 Shadow mode foothold

The shadow mode foothold allows an attacker to inject and execute arbitrary code. The shadow mode foothold works by monitoring data values as they pass between the cache and the pipeline, and installs an invisible firmware within the processor when a specific value triggers the attack. When this firmware runs, it runs with full processor privileges, it can gain control of the processor at any time, and it remains hidden from software running on the system. To provide storage for exploit instructions and data, this foothold reserves blocks in the instruction and data caches for storing injected instructions and data. The shadow mode foothold is triggered with a sequence of bytes and the shadow mode foothold interprets the bytes following the trigger sequence as commands and machine instructions.

To evaluate BlueChip, we implement the "bootstrap trigger" portion of

the shadow mode foothold. The bootstrap trigger waits for a predetermined value to be stored to the data cache, and asserts a processor exception that transfers control to a hard-coded "bootstrap code" that resides within the processor cache. Our implementation includes the "bootstrap trigger" and asserts a processor exception, but omits the "bootstrap code" portion of the foothold. As a result, we are unable to implement full system attacks using our version of the foothold, but it does give us enough of the functionality of the shadow mode foothold to enable us to evaluate our defense because removing the "bootstrap trigger" disables the attack.

5.7 BlueChip prototype

To experimentally verify the BlueChip approach, we prototyped the hardware, the software, and the design-time UCI analysis algorithm.

We based our hardware implementation on the Leon3 processor [88] design. Our prototype is fully synthesizable and runs on an FPGA development board that includes a Virtex 5 FPGA, CompactFlash, Ethernet, USB, VGA, PS/2, and RS-232 ports. The Leon3 processor implements the SPARC v8 instruction set [105] and our configuration uses eight register windows, a 16 KB instruction cache, a 64 KB data cache, includes an MMU, and runs at 100 MHz, which is the maximum clock rate we are able to achieve for the unmodified Leon3 design, for our target FPGA. For the software, we use a SPARC port of the Linux 2.6.21.1 kernel on our FPGA board and we install a full Slackware distribution on our system. By evaluating BlueChip on an FPGA development board and by using commodity software, we have a realistic environment for evaluating our hardware modifications and accompanying software systems.

To insert our BlueChip hardware modifications, we wrote tools that take as input data-flow pairs generated by our UCI implementation and automatically transforms the Leon3 VHDL source code to replace suspicious circuits and add exception delivery logic. Our tool is mostly hardware-implementation agnostic and should work on a wide range of hardware designs automatically. The only hardware implementation specific portions of our tool are for connecting the BlueChip logic to the Leon3 exception handling stage.

For our UCI implementation, we wrote a VHDL compiler front end in Java which generates a data-flow graph from arbitrary HDL source code, determines all possible pairs of edges in the data-flow graph, then uses TCL to automatically drive ModelSim, running the simulation and removing pairs that are violated during testing. The last stage of our UCI implementation performs a source-to-source transformation using the original VHDL design and remaining data-flow pairs to generate a design with BlueChip hardware.

Our BlueChip software runs as an exception handler within the Linux kernel. The BlueChip emulation code is written in C and it can emulate all non-privileged SPARC instructions and most of the privileged operations of the Leon3 SPARC implementation. Because SPARC is a reduced instruction set computer (RISC), we implemented our emulator using only 1759 lines of code and it took us about a week to implement our instruction emulation routines.

We identify suspicious hardware using three sets of tests: the basic test suite included with the Leon3 distribution, SPARC certification test cases from SPARC International, and five additional test cases to test portions of the instruction set specification that are uncovered by the basic Leon3 and SPARC certification test cases. To identify suspicious logic, we simulate the HDL using ModelSim version 6.5 and perform the UCI analysis on the simulation results. Our analysis focuses on the integer unit (*i.e.*,, the core pipeline) of the Leon3 processor.

5.8 BlueChip evaluation

This section describes our evaluation of BlueChip. In our evaluation, we measure BlueChip's: (1) ability to stop attacks, (2) ability to successfully emulate instructions that used hardware removed by BlueChip, and (3) hardware and software overheads.

5.8.1 Methodology

To evaluate BlueChip's ability to prevent and recover from attacks, we wrote software that activates the malicious hardware described in Section 5.7. We designed the software to activate and exploit the low-level footholds imple-

mented by our attacks, and tested to see if these out-of-spec abstractions were rendered powerless and if the system could make post attack progress.

To identify suspicious circuits, we used three different sets of hardware design verification tests. First, we used the Gaisler test suite that comes bundled with the Leon3 hardware's HDL code. These test cases use ISA-level instructions to test both the processor core and peripheral (*i.e.*,, outside the processor core) units like the caches, memory management unit, and system-on-chip units such as the UART. Second, we used the official SPARC verification tests from SPARC International, which are used to ensure compatibility with the SPARC instruction set. These test cases are designed to confirm that a processor implements the instructions and architecturally visible states needed to be considered a SPARC processor, but they are not intended to be a complete design verification suite. Third, we created a small set of custom hardware test cases to improve design coverage, closer to what is common in a production environment. The custom test cases cover gaps in the Gaisler test cases and exercises instructions that Leon3 supports, but are optional in the SPARC ISA specification (*e.g.*,, floating-point operations).

To measure execution overhead, we used three workloads that stressed different parts of the system: `wget` fetches an HTML document from the Web and represents a network bound workload, `make` compiles portions of the `ntpdate` application and stresses the interaction between kernel and user modes, and `djpeg` decompresses a 1MB jpeg image as a representative of a compute-bound workload. To address variability in the measurements, reported execution time results are the average of 100 executions of each workload relative to an uninstrumented base hardware configuration. All of our overhead experiments have a 95% confidence interval of less than 1% of the average execution time.

5.8.2 Does BlueChip prevent the attacks?

There are two goals for BlueChip when aiming to defend against malicious hardware. The first and most important goal is to prevent attacks from influencing the state of the system. The second goal is for the system to recover, allowing non-malicious programs to make progress after an attempted attack.

The results in Figure 5.8 show that BlueChip successfully prevents all

Attack	Prevent	Recover
Privilege Escalation	√	√
Memory Redirection	√	
Shadow Mode	√	√

Figure 5.8: *BlueChip attack prevention and recovery.*

three attacks, meeting the primary goal for success. BlueChip meets the secondary goal of recovery for two of the three attacks, but it fails to recover from attempted activations of the memory redirection attack. In this case, the attack is prevented, but software emulation is unable to make forward progress. Upon further examination, we found that the Leon3's built-in pipeline recovery mechanism was insufficient to clear the attack's internal state. This lack of progress is due to the attack circuit ignoring the Leon3 control signal that resets registers on pipeline flushes, thus making some attack states persist even after pipeline flushes. This situation causes the BlueChip hardware to repeatedly trap to software, thus blocking forward progress, but preventing the attack. Our analysis indicates that augmenting Leon3's existing recovery mechanism to provide additional state recovery would allow BlueChip to recover from this attack as well.

5.8.3 Is software emulation successful?

BlueChip justifies its aggressive identification and removal of suspicious circuits by relying on software to emulate any mistakenly removed functionality. Thus, BlueChip will trigger spurious exceptions (*i.e.*,, those exceptions that result from removal of logic mistakenly identified as malicious). In our experiments, all of the benchmarks execute correctly, indicating BlueChip correctly recovers from the spurious BlueChip exceptions that occurred in these workloads.

Figure 5.9 shows the average rate of BlueChip exceptions for each benchmark. Even in the worst case, where a BlueChip exception occurs every 20ms on average, the frequency is far less than the operating system's timer interrupt frequency. The rate of BlueChip exceptions is low enough to allow for complex software handlers without sacrificing performance.

Figure 5.10 shows the experimental data used to quantify the effectiveness

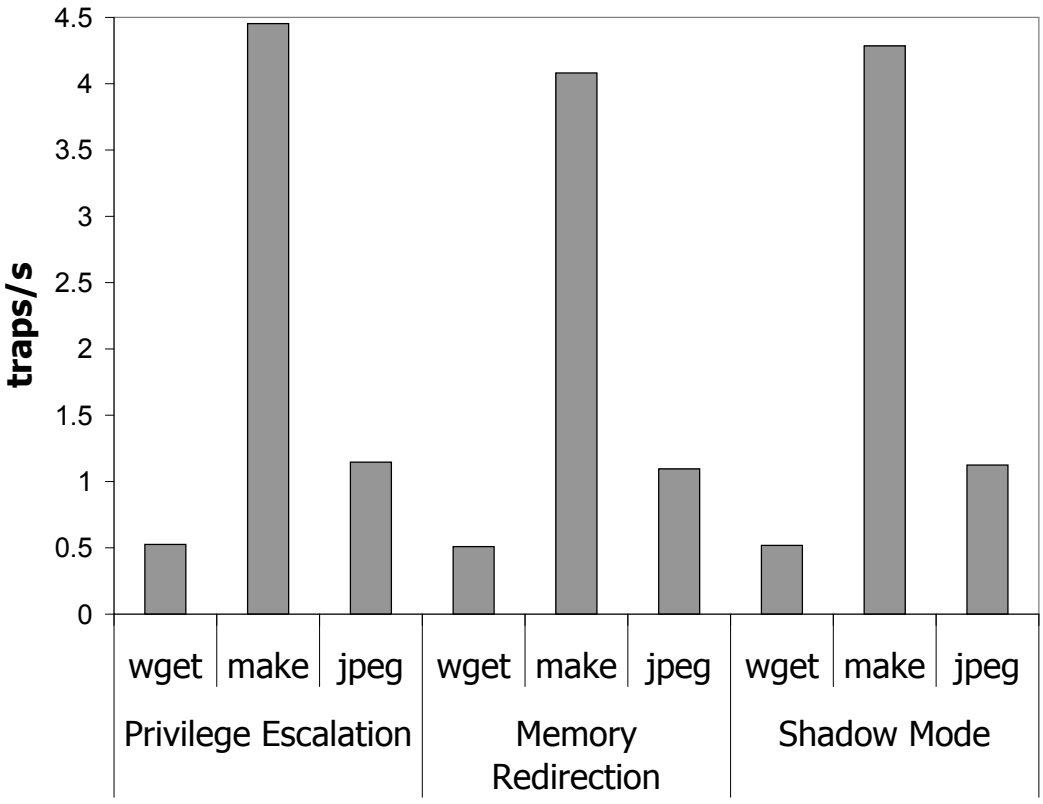


Figure 5.9: *BlueChip software invocation frequencies.*

---

of UCI. This figure shows the number of suspicious pairs remaining after each stage of testing. The misidentified pairs, which are all false positives for our tests, are the number of suspicious pairs minus the number of attack pairs detected. These false positive pairs can manifest themselves as spurious BlueChip exceptions during runtime. The number of pairs remaining after testing affects the likelihood of seeing spurious BlueChip exceptions, with fewer pairs generally leading to less frequent traps. Even though some of the remaining pairs result in spurious exceptions, the instruction-level emulation provided by BlueChip software hides this behavior from the rest of the system, thus allowing unmodified applications to execute unaware that they are running on BlueChip hardware.

The discrepancy in the number of traps experienced by each benchmark is also worth noting. The `make` benchmark experiences the most traps, by almost an order of magnitude. Looking at the UCI pairs that fire during testing, and looking at the type of workload `make` creates, the higher rate of traps comes from interactions between user and kernel modes. This happens

	Baseline		Gaisler Tests		+SPARC Tests		+Custom Tests	
	Attack	**All**	**Attack**	**All**	**Attack**	**All**	**Attack**	**All**
Privilege Escalation	2	3046	1	103	1	87	1	39
Memory Redirection	54	3098	8	110	8	94	8	46
Shadow Mode	8	3051	1	103	1	87	1	39

Figure 5.10: *UCI dataflow pairs. This figure shows how many dataflow pairs UCI identifies as suspicious for the Gaisler test suite, the SPARC verification test suite, and our custom test cases, cumulatively. The data shows the number of dataflow pairs identified total, and shows how many of these pairs are from attack circuits.*

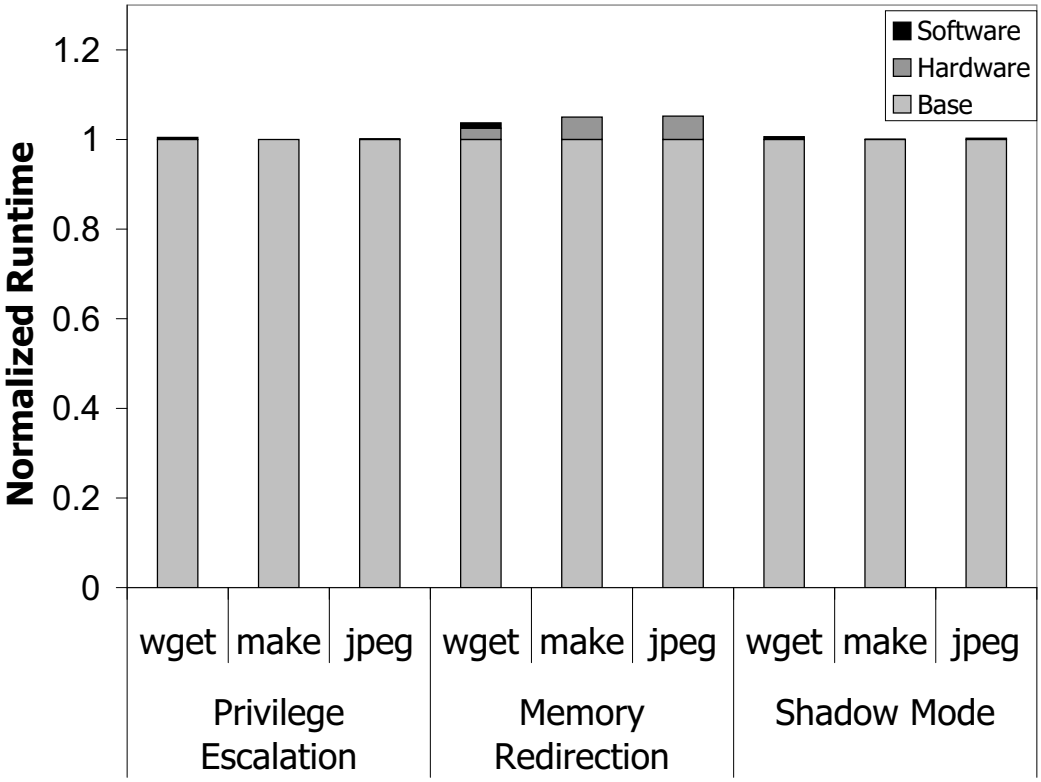


Figure 5.11: *Application runtime overheads for BlueChip systems.*

more often in `make` than the other benchmarks, as `make` creates a new process for each compilation. More in-depth tracing of the remaining UCI pairs reveals that many pairs surround the interaction between kernel mode and user mode. Because UCI is inherently based on design verification tests, this perhaps indicates the parts of hardware least tested in our three test suites. Conversely, the relatively small rate of BlueChip exceptions experienced by `wget` is due to its I/O (network) bound workload. Most of the time is spent waiting for packets, which apparently does not violate any of the UCI pairs remaining after testing.

5.8.4 Is BlueChip's runtime overhead low?

Although BlueChip successfully executes our benchmark workloads, frequent spurious exceptions have the potential to significantly impact system performance. Furthermore, BlueChip's hardware transformation could impact the hardware's operating frequency.

Attack	Power (W)	Area (Luts)	Freq. (MHz)
Privilege Escalation	0.41%	1.38%	0.00%
Memory Redirection	0.47%	1.19%	5.00%
Shadow Mode	0.29%	0.31%	0.00%

Figure 5.12: *BlueChip hardware overheads for each of our attacks.*

Figure 5.11 shows the normalized breakdown of runtime overhead experienced by the benchmarks running on a BlueChip system versus an unprotected system. The runtime overhead from the software portion of BlueChip is just 0.3% on average. The software overhead comes from handling spurious BlueChip exceptions, primarily from just two of the UCI pairs. The average overhead from the hardware portions of BlueChip is approximately 1.4%.

Figure 5.12 shows the relative cost of BlueChip in terms of power, device area, and maximum operating frequency. The hardware overhead in terms of area averages less than 1% of the entire design. Even though BlueChip needs hardware to monitor the remaining pairs, much of the hardware already exists and BlueChip just taps the pre-existing signals. The majority of the area overhead comes from the comparisons used to determine if a BlueChip exception is required given the state of the pipeline. To reduce BlueChip's impact on maximum frequency, these comparisons happen in parallel and BlueChip's exception generation uses a tree of logical-OR operations. In fact, for the privilege escalation and shadow mode versions of BlueChip, there is no measurable impact on maximum frequency, indicating that UCI pair checking hardware is typically off the critical path. For the memory redirection attack hardware design, some of the pair checking logic is placed on the memory path, which is the critical path for this design and target device. In this case, the maximum frequency is reduced by five percent. Consistent with the small amount of additional hardware, the power overhead averages less than 0.5%.

5.9 Conclusions

BlueChip neutralizes malicious hardware introduced at design time by identifying and removing suspicious hardware during the design verification phase,

while using software at runtime to emulate hardware instructions to avoid erroneously removed circuitry.

Experiments indicate that BlueChip is successful at identifying and preventing attacks while allowing non-malicious executions to make progress. Our malicious circuit identification algorithm, UCI, relies on the attempts to hide functionality to identify candidate circuits for removal. BlueChip replaces circuits identified by UCI with exception logic, which initiates a trap to software. The BlueChip software emulates instructions to detour around the removed hardware, allowing the system to attempt to make forward progress. Measurements taken with the attacks inserted show that such exceptions are infrequent when running a commodity operating system using traditional applications.

In summary, these results show that addressing the malicious insider problem for hardware design is both possible and worthwhile, and that approaches can be cost effective and practical.

Chapter 6

Future work

The reference implementations presented in this dissertation show that it is possible to detect a range of processor imperfections and help software make consistent forward progress. Experiments with the reference implementations show that detection and recovery are practical. Experiments also expose limitations with the current incarnations, motivating future work.

The major limitation of BlueChip is its inability to make guarantees—or any concrete claims—about the likelihood of recovery. As discussed previously, the likelihood of recovery is dependent on the functional redundancy of the processor with respect to the imperfection. Processor designers currently have no way to systematically adjust functional redundancy or focus verification effort to increase the chances of recovery. Future work makes BlueChip more palatable to processor designers by creating a feedback loop that not only provides information on the bounds of recovery, but leads towards a formally verified recovery primitive.

Another major limitation, exposed by the experiments with Erratacator, is that it is not practical to detect arbitrary processor imperfections (which number into the hundreds [106, 3, 107]), given the current state of research. Future work will look to guard only a subset of the processor from the ISA level, which is more regular and less complex than the hardware or microarchitectural level. This way, processor designers can determine the most critical parts of the interface to software (*e.g.,* privileged state or recovery primitive support) and protect it, relying on lower overhead methods of protection for non-critical functionality.

The end goal of the proposed future research is the creation of a automatically generated, formally verified, general purpose, and execution-protected recovery primitive. The main advantage of the proposed recovery primitive being that it is essentially free as it reuses existing processor functionality—no added area, power, or complexity—and adds no run time overhead to

software.

6.1 Guaranteeing forward progress

BlueChip, as it stands, is a proof-of-concept, not ready for adoption by commercial processor designers. First, the recovery routines used in the reference implementations were either targeted to the imperfection, or used the general-purpose instruction set simulator. The targeted recovery routines provide a higher assurance of recovery at the expense of the labor required to manually inspect the HDL and code a custom recovery routine in assembly. It is impractical to expect processor designers to do this and not make an error. Second, with the general-purpose simulator, you have no sense of the average recovery time: the simulator just attempts different recoding algorithms, until one works, or until failure is reached. This leads to the third limitation, that there is no way for processor designers to know what the likelihood of recovery is. These limitations mean that processor designers cannot make intelligent tradeoffs: Adding functional redundancy to increase the likelihood of recovery or to decrease the average recovery time. Or, manipulating functionality to enable complete formal verification.

To address these limitations, making BlueChip ready for use in commercial processors, we propose enhancements to BlueChip that create a feedback loop, informing processor designers how the processor's functionality impacts formal verification, likelihood of recovery, and average recovery time. The proposal consists of two parts: identifying the functionality amenable to formal verification and identifying the functionality sufficient for forward progress. By combining the two parts, we can identify a subset of processor functionality that enables complete formal verification and is also sufficient to guarantee software's forward progress, allowing BlueChip to become an almost no cost recovery primitive.

The idea of repurposing existing processor functionality for recovery differs from previous hardware-based approaches in that the proposed system requires no other component to guarantee software's forward progress. Previous proposals require additional, fully-functional processing units: typically, a fault-free system [108] or a simplified processor tightly coupled to the main processor [10]. In remote approaches, recovery times are on the order of sec-

onds and users must pay for the remote system even though they rarely use it. In local approaches, while recovery times approach zero, the added processors increase the chip's area and power draw. The increased area makes chips more expensive to produce, while the added power draw makes them more expensive to operate. A final drawback is that both approaches strictly increase the complexity of the system, increasing the likelihood of creating new—potentially unrecoverable—bugs. With all these penalties, users and processor designers end-up paying more for a less productive system.

The proposed work involves answering three questions: (1) What is the relationship between functionality and verifiability? (2) What is the relationship between functionality and the likelihood of forward progress? (3) Is it possible to create a feedback mechanism that helps designers tradeoff functionality for verifiability or recoverability? With these questions answered, processor designers can replace the added complexity and overheads of traditional hardware-only functional redundancy with a systematic repurposing of pre-existing processor functionality that is useful in the common case.

6.1.1 Functionality vs. verifiability

The idea here is to explore if certain parts of a processor's functionality are more difficult to verify than others. This differs from previous research into the formal verification of processors in that previous proposals took a microarchitectural view, making processors more amenable to verification by ignoring their internal workings, *e.g.,* pipelining [74]. The problem is that processors do not run with their pipelines disabled. We approach the problem from an instruction perspective: looking at which instructions we can eliminate support for to decrease verification effort. Taken to the extreme, the question becomes, "Is it possible to verify this processor if it only supports one instruction?" If it is possible we observe the effects of adding new instructions. This exposes to processor designers a fresh opportunity to tradeoff known correct functionality for verification effort.

6.1.2 Functionality vs. forward progress

The idea here is to identify small subsets of a processor's instruction-level functionality, which have the same expressive power as the entire ISA. In the ideal case, we would reduce the subsets down to a kernel of functionality that is not both sufficient and necessary to make forward progress guarantees. Proving that the kernel of functionality is both necessary and sufficient needs to be done both theoretically, by showing that the ISA-level effects from any instruction in the ISA can be reproduced using only instructions in the subset and that removing any instruction from the subset breaks the previous statement, and realistically, by showing the no limitations due to the physical implementation cause a loss of expressiveness. An example of a limitation imposed by the implementation that is not covered in the ISA is the Byte Bus on the OR1200. Accesses to the Byte Bus must use the byte-wide load and store instructions; anything else is an error, resulting in an exception. This physical limitation means that any kernel of functionality must include the byte-wide load and store instructions.

Once we find and prove equivalent the kernel of functionality for the processor, other questions arise. Is it possible and beneficial (*e.g.,* easier to formally verify) to limit the functionality of the instructions that make up the kernel? What is the overhead of recoding each instruction not in the kernel using only instructions in the kernel? How does recovery overhead and formal verification overhead change as we add more instructions to the kernel?

6.1.3 Implications

When processor designers can understand the formal verification and recovery consequences of adding or removing functionality to the ISA, they will have the power to systematically design instruction sets with verification and use in mind. It would be less risky for designers to add new functionality, as they could assure themselves that the recovery subset of processor functionality could take over the task in the event of a imperfection. By fabricating the subset of processor functionality in a larger process or using more robust circuit elements, processors would be more resilient to single-event-upsets or attacks that involve running the processor outside the specified operat-

ing conditions. With the proposed modifications, BlueChip could act as a minimal hardware TCB.

6.2 Enforcing execution

The ISA specification is the contract between the processor and the software that wants to run on the processor. The ISA sets the expectations of software. Software must blindly assume that the processor correctly implements the ISA, given all possible ISA-level states, referred to as states, and ISA-level events (*i.e.,* instructions, exceptions, and interrupts), referred to as events.

The problem is that, being complex, processors fail to correctly implement the ISA. Certain combinations of states and events activate the processor imperfections, contaminating the state of software. This is especially worrisome in security-critical systems, where evidence shows that contamination via processor imperfections has already lead to vulnerabilities in otherwise correctly functioning software [**refs**].

The goal of this work is to identify the security-critical subset of a generalized processor's state, then to design and build a system that validates the values of the security-critical subset of state at run time. With such a system in place, secure software will know that, no matter what, it can rely on the processor's abstractions for maintaining privilege, data privacy, and data integrity.

The main research questions that need to be answered to build such a system are: (1) How to find a security-critical subset of processor state, (2) Should the protections be hardware-implemented or software implemented, (3) What is the best way to validate state, (4) What to do in the event of contamination, and (5) How to gauge the efficacy of the added protections.

6.2.1 Identifying a security-critical subset

We create a testbed for the system by gathering all security-related errata from the top three commercial processors released (by units sold) within the last ten years. From the list of security-related errata, we create classes of vulnerabilities, which we then implement in our own processor. The re-implementations have the same contamination of the original, but with the

freedom to have any trigger: we aim to attack our own defenses. Any attack that we cannot code an assertion to detect signals a hole in our system.

Choosing this subset of processor functionality makes dynamic verification practical by reducing the range of functionality required by the verification engine; remember that the goal is to avoid re-implementing the processor.

Dynamically verifying the entire processor is impractically because it requires building another processor and working it into the system; Diva [**refs**]is an example of this. The idea is to identify a subset of processor functionality that is critical to software, but requires re-implementing a small subset of process functionality.

6.2.2 Hardware or software protections

The verification needs to be done in hardware as verifying the processor in software—without risking false negatives—requires stopping the processor after every instruction and event and running a complex verification algorithm. In addition, software has limited visibility into the events inside a processor. As discussed in Section 6.2.6, software is best suited at monitoring general-purpose state in a practical manner.

6.2.3 Detecting contamination

The role of the detectors is to detect, at run time, any out-of-specification updates to the protected subset of the processor's ISA-level state. This requires both hooks into the processor's ISA-level state and a way to encode the ISA in hardware, without implementing another processor.

By hooking into the processor's state, the detectors are able to track the value of each protected state element. To accomplish this processor designers need to specify which HDL variable names comprise ISA-level state elements. Experience with the two reference designs presented in this dissertation shows that the difficulty of identifying all of the ISA-level state varies depending on the language and implementation style used by the processor's designers. Once the processor designers identify all of the ISA state, the detectors will be able to track all changes to it, down to the clock cycle.

Given the processors state, the detectors need a way to determine if the

current value is outside of what you would expect give the previous state, the instruction, and the ISA specification. The detectors must check not only updates to the protected state, but must check for cases of a missing update. For example, a privilege escalation attack has the processor go from user to supervisor mode in an out-of-specification way, but it is also possible to create a privilege anti-de-escalation attack, where the processor remains in supervisor mode when it should have gone to user mode (*e.g.,* a return from exception).

We map these two possible attack vectors—a surreptitious change and a disregarded change—as violations of two different implications: If protected state X changes, then event Y must have occurred. If event Y occurred, then protected state X must have changed. It is possible to encode the specification as a set of implications. Going back to the privilege example, the specification says that privilege can only escalate (P_esc) in the event of a reset (rst) or an exception/interrupt/system call (eisc_enter). The specification also says that upon return from a exception/interrupt/system call (eisc_exit), the privilege bit (P) is set to what it was before the call (P_PRE_EISC). After encoding these rules as implications we get, P_esc $\longrightarrow$ (rst or eisc_enter) and eisc_exit $\longrightarrow$ (P == P_PRE_EISC). This motivates the use of assertions to verify that these implications hold.

The challenges are identifying a good set of ISA-level state that enables expressive assertions and set of operations that supports the required comparisons of state and events.

6.2.4 Recovering from contamination

It is not sufficient to stop at detecting state contamination, because the contaminated state may not be flushed away and software is likely to trigger the imperfection again, locking the system. These issues necessitate the addition of a recovery module that takes control after a detection, clears any remaining contamination, makes it unlikely that software will re-trigger the imperfection, and finally passes full control of the processor back to software.

The first role of the recovery module is to remove any state contamination due to the imperfection. The ideal is to rollback the entire ISA-level state of the process to what it was before the imperfection was triggered. The ideal is

not practical as it requires full-system checkpointing support. Checkpointing just the security-critical state is not sufficient, because even though only the security-critical subset of state is protected (*i.e.,* a recovery firmware activation is only due to a detected contamination of the security-critical state) a complete decontamination might require modifications to general-purpose state. For example, if an ADD instruction somehow clobbers any array of general-purpose registers, while setting the privilege bit to Supervisor mode, it would be prudent to clean-up the entire mess, not just set the privilege bit back to User mode. Experiments with the reference implementation will shed more light on the issue.

If we just look at decontaminating security-critical state, two options arise: setting the state to an obvious value given the type of imperfection detected (*e.g.,* privilege bit to 0 (User mode) after an out-of-spec privilege escalation attempt) and setting the state to a known safe value known safe value (*e.g.,* make page as invalid after a out-of-spec modification of the hardware page table registers). Note that these two cases are not exhaustive.

The second role of the recovery module is to help software make forward progress, with little chance of re-triggering the imperfection. For this, we rely on the previously proposed general-purpose instruction set simulator, because it is formally guaranteed to push software state forward and much more lightweight.

The simulator is not enough to prevent an the re-triggering of an imperfection. Making re-triggering unlikely requires resetting the imperfections trigger sequence back to its starting state. Previous work shows that a pipeline flush is not capable of guaranteeing this [45]. To guarantee a reset of the trigger sequence requires resetting the hardware state.

6.2.5 Testing the protections

To verify that the proposed protections work, we implement an attack that represents each class of processor imperfection found in Section 6.2.1. The attacks have the same spirit of the effect listed in the errata documents, but are not exact reproductions since our goal is to expose weaknesses in the design and since the processors differ greatly. After adding the attacks to the reference implementation and developing a code sequence that triggers

the imperfection, we run benchmarks intermixed with the trigger sequence, checking for unhandled state contamination.

Using the attack testbed not only exposes any holes in the approach, both gives an idea of the real-world overheads of using such a defense.

6.2.6 Limitations

There are two limitations to the proposed mechanism, one, it will not detect contamination of the general-purpose subset of a processor's state. This is by design, as protecting all of a processor's state requires an artifact with the same complexity as the first, which is not practical. Also there are alternative software-based mechanisms that exist for validating general purpose state, *e.g.,* redundant execution [34, 41, 109], which are made practical by applying them on a per-process basis. Note that these techniques cannot verify the security-critical subset of the processor, thus they require the proposed protections.

Two, it is possible for a processor imperfection to contaminate the processor's general-purpose state and have that contamination spread to protected state. An example of this is a corrupted XOR operation that the occurs when the operating system is computing page protections. The operating system will update the paging hardware with an incorrect value, but since the operating system has the privilege required to update the paging hardware, the processor behaves within specification in performing the update. We refer to this as indirect contamination.

Indirect contamination is directly controlled by software through the instructions it issues. Any slight modification to the triggering instruction sequence is likely to avoid the contamination. Considering the fixed nature of hardware-based attacks and the ease and frequency of software patches, we feel that indirect contamination is *not* a serious threat.

6.2.7 Implications

If successful, software could manage its exposure to potential processor imperfections, no longer needing to blindly trust the processor to correctly implement abstractions it cannot validate. Processor designers could also forgo

the often ignored errata documents in favor of providing sets of assertions that actually fix the problem.

6.3 Conclusion

With feedback on the bounds of recovery, processor designers can concentrate verification effort on a the subset of the processor required for recovery.

With the ability to enforce a subset of the ISA, software can be confident that the portion of the hardware interface that it relies on the most is protected.

Combining guarded execution and a recovery primitive expands the utility of the recovery primitive beyond processor imperfections. One could imagine the value of such a combination in environments with transient faults. This includes attack scenarios where the processor is forced to run outside of its specified operating environment.

Chapter 7

Conclusion

Experiments with both use case implementations show that, in conjunction with hardware-implemented detectors, SoftPatch is both general purpose and practical. The detectors are acceptable to hardware designers, because (1) they do not interact with the processor outside of the exception support logic, maintains the original design's complexity and (2) they incur less than 10% hardware area overhead for the processor bug use case and less than 2% for the malicious circuit use case. SoftPatch is acceptable to software developers and users because it quietly maintains the perfect-processor abstraction while maintaining overheads of less than 10% in real-world tests in both use cases.

Experiences from building the two use case implementations demonstrate the general purpose nature of SoftPatch. Our observation is that the design of the detector is independent from the design and verification of the recovery firmware. By formally verifying the equivalence of the many recovery routines, SoftPatch adapts to different imperfections by employing different recovery routines in hopes that one will successfully route around the imperfection. Only when it becomes important to reduce the execution time of the recovery routines does knowledge of the imperfection become important.

In the future, we see a usage scenario where processor designers intentionally under-verify processor functionality, guarding the under-verified functionality with dynamically configurable detectors that utilize SoftPatch in the event that post-deployment verification discovers a bug. This allows the processor designer to explore the trade-off between pre-deployment verification, detector overhead, and software run time overhead due to recovery.

This work shows that it is possible and practical to simulate in software what is missing or defective in hardware without understanding the imperfection or imposing on the software.

Bibliography

[1] S. Systems, "OpenSPARC T2 HW 1.3 Released," http://www.opensparc.net/whats-new/2009-q3/opensparc-t2-hw-13-released.html, July 2009.

[2] Jon Stokes, "Two billion-transistor beasts: Power7 and niagara 3," http://arstechnica.com/business/news/2010/02/two-billion-transistor-beasts-power7-and-niagara-3.ars.

[3] Intel, "Intel core 2 extreme processor x6800 and intel core 2 duo desktop processor e6000 and e4000 sequence – specification update," Intel Corporation, Tech. Rep., May 2008.

[4] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou, "Designing and implementing malicious hardware," in *Proceedings of the First USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*.

[5] S. Shebs, "Gdb tracepoints, redux," in *Proceedings of the GCC Developer's Summit*.

[6] MIPS, "MIPS technologies. MIPS R4000PC/SC errata, processor rev. 2.2 and 3.0," May 1994.

[7] B. Yee, D. Sehr, G. Dardyk, J. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," in *Symposium on Security and Privacy*, pp. 79 –93.

[8] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, "Understanding the propagation of hard errors to software and implications for resilient system design," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 265–276.

[9] K. Constantinides, O. Mutlu, and T. Austin, "Online design bug detection: Rtl analysis, flexible mechanisms, and evaluation," in *IEEE/ACM International Symposium on Microarchitecture*, pp. 282–293.

[10] T. M. Austin, "Diva: a reliable substrate for deep submicron microarchitecture design," in *ACM/IEEE International Symposium on Microarchitecture*, pp. 196–207.

[11] B. Gassend, D. Clarke, M. van Dijk, and S. Devadas, "Silicon physical random functions," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pp. 148–160.

[12] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar, "Trojan detection using IC fingerprinting," in *IEEE Symposium on Security and Privacy*.

[13] D. Sokolov, J. Murphy, A. Bystrov, and A. Yakovlev, "Design and analysis of dual-rail circuits for security applications," *IEEE Trans. Comput.*, vol. 54, no. 4, pp. 449–460, 2005.

[14] K. Tiri and I. Verbauwhede, "Design method for constant power consumption of differential logic circuits," in *DATE '05: Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 628–633.

[15] J. Kumagai, "Chip detectives," *IEEE Spectr.*, vol. 37, no. 11, pp. 43–49, 2000.

[16] T. Huffmire, B. Brotherton, G. Wang, T. Sherwood, R. Kastner, T. Levin, T. Nguyen, and C. Irvine, "Moats and drawbridges: An isolation primitive for reconfigurable hardware based systems," in *Symposium on Security and Privacy*, pp. 281–295.

[17] C. Sturton, M. Hicks, D. Wagner, and S. T. King, "Defeating uci: Building stealthy and malicious hardware," in *Symposium on Security and Privacy*.

[18] G. Wang, Y. Zhang, Q. Shi, and X. Ma, "A self-learning structure to improve the efficiency of bluechip software," in *International Conference on Intelligence Science and Information Engineering*, pp. 393–396.

[19] L.-W. Kim, J. D. Villasenor, and c. K. Koç, "A trojan-resistant system-on-chip bus architecture," in *conference on Military communications*, pp. 2452–2457.

[20] S. Narayanasamy, B. Carneal, and B. Calder, "Patching processor design errors," in *IEEE International Conference on Computer Design*, pp. 491–498.

[21] S. R. Sarangi, A. Tiwari, and J. Torrellas, "Phoenix: Detecting and recovering from permanent processor design bugs with programmable hardware," in *IEEE/ACM International Symposium on Microarchitecture*, pp. 26–37.

[22] B. Lampson, M. Abadi, M. Burrows, and E. Wobber, "Authentication in distributed systems: theory and practice," *ACM Transactions on Computer Systems*, vol. 10, no. 4, pp. 265–310, November 1992.

[23] J. D. Rolt, G. D. Natale, M.-L. Flottes, and B. Rouzeyre, "New security threats against chips containing scan chain structures," in *International Symposium on Hardware-Oriented Security and Trust*, pp. 105–110.

[24] R. S. Chakraborty, S. Narasimhan, and S. Bhunia, "Hardware trojan: Threats and emerging solutions," in *International High Level Design Validation and Test Workshop*, pp. 166–171.

[25] A. Pellegrini, V. Bertacco, and T. Austin, "Fault-based attack of rsa authentication," in *Conference on Design, Automation and Test in Europe*, pp. 855–860.

[26] R. Lveugle, "Early analysis of fault-based attack effects in secure circuits," *Transactions on Computers*, vol. 56, no. 10, pp. 1431–1434, October 2007.

[27] S. Adee, "The hunt for the kill switch," *IEEE Spectrum*, vol. 45, no. 5, pp. 34–39, 2008.

[28] M. Banga and M. S. Hsiao, "A region based approach for the identification of hardware trojans," in *International Workshop on Hardware-Oriented Security and Trust*, pp. 40–47.

[29] R. Rad, J. Plusquellic, and M. Tehranipoor, "A sensitivity analysis of power signal methods for detecting hardware trojans under real process and environmental conditions," *Transactions on Very Large Scale Integration Systems*, vol. 18, no. 12, pp. 1735–1744, December 2010.

[30] J. Li and J. Lach, "At-speed delay characterization for ic authentication and trojan horse detection," in *International Workshop on Hardware-Oriented Security and Trust*, pp. 8–14.

[31] M. Banga, M. Chandrasekar, L. Fang, and M. S. Hsiao, "Guided test generation for isolation and detection of embedded trojans in ics," in *Great Lakes symposium on VLSI*, pp. 363–366.

[32] B. Parhami, "Defect, fault, error, ..., or failure?" *IEEE Transactions on Reliability*, vol. 46, no. 4, pp. 450–451, December 1997.

[33] D. Lammers, "The era of error-tolerant computing," November 2010, IEEE Spectrum. [Online]. Available: http://spectrum.ieee.org/semiconductors/processors/the-era-of-errortolerant-computing

[34] J. Chang, G. A. Reis, and D. I. August, "Automatic instruction-level software-only recovery," in *International Conference on Dependable Systems and Networks*, pp. 83–92.

[35] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood, "Safetynet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery," in *International Symposium on Computer Architecture*, pp. 123–134.

[36] I. Wagner and V. Bertacco, "Caspar: hardware patching for multi-core processors," in *Conference on Design, Automation and Test in Europe*, European Design and Automation Association, pp. 658–663.

[37] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An architectural framework for software recovery of hardware faults," in *International Symposium on Computer Architecture*, pp. 497–508.

[38] A. Meixner, M. E. Bauer, and D. Sorin, "Argus: Low-cost, comprehensive error detection in simple cores," in *IEEE/ACM International Symposium on Microarchitecture*, pp. 210–222.

[39] P. M. Wells, K. Chakraborty, and G. S. Sohi, "Adapting to intermittent faults in multicore systems," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 255–264.

[40] A. Meixner and D. J. Sorin, "Detouring: Translating software to circumvent hard faults in simple cores," in *International Conference on Dependable Systems and Networks*, pp. 80–89.

[41] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *International Symposium on Code Generation and Optimization*, pp. 243–254.

[42] J. F. F. Sellers, M.-Y. Hsiao, and L. W. Bearnson, *Error Detecting Logic for Digital Computers*. McGraw-Hill, 1968.

[43] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco, "Software-based online detection of hardware defects: Mechanisms, architectural support, and evaluation," in *IEEE/ACM International Symposium on Microarchitecture*, pp. 97–108.

[44] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin, "Ultra low-cost defect protection for microprocessor pipelines," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 73–82.

[45] M. Hicks, M. Finnicum, S. T. King, M. M. K. Martin, and J. M. Smith, "Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically," in *IEEE Symposium on Security and Privacy*, pp. 159–172.

[46] I. Hadžić, S. Udani, and J. M. Smith, "Fpga viruses," in *nternational Workshop on Field-Programmable Logic and Applications*.

[47] Y. Jin, N. Kupp, and Y. Makris, "Experiences in hardware trojan design and implementation," *Hardware-Oriented Security and Trust, IEEE International Workshop on*, vol. 0, pp. 50–57, 2009.

[48] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Symposium on Operating Systems Design and Implementation*.

[49] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: automatically generating inputs of death," in *CCS '06: Proceedings of the 13th ACM Conference on Computer and Communications Security*, pp. 322–335.

[50] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu, "Symbolic model checking using SAT procedures instead of BDDs," in *Proc. ACM/IEEE Annual Design Automation Conference*, pp. 317–320.

[51] H. Chen, D. Dean, and D. Wagner, "Model checking one million lines of C code," in *Network and Distributed System Security Symposium*, pp. 171–185.

[52] J. Yang, P. Twohey, D. Engler, and M. Musuvathi, "Using model checking to find serious file system errors," in *OSDI'04: Proceedings of the 6th Symposium on Operating Systems Design & Implementation*, pp. 273–288.

[53] A. C. Myers and B. Liskov, "A decentralized model for information flow control," in *Symposium on Operating Systems Principles*.

[54] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers, "Untrusted hosts and confidentiality: secure program partitioning," in *Proceedings of the 2001 Symposium on Operating Systems Principles*.

[55] A. Waksman and S. Sethumadhavan, "Tamper evident microprocessors," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*.

[56] L. Lamport, R. E. Shostak, and M. C. Pease, "The byzantine generals problem." *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, 1982.

[57] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault-tolerance," in *Symposium on Operating Systems Principles*, pp. 1–11.

[58] U. Science, *Defense Science Board Task Force on High Performance Microchip Supply*. General Books, 2005. [Online]. Available: http://books.google.com/books?id=cikBywAACAAJ

[59] L. C. Heller and M. S. Farrell, "Millicode in an ibm zseries processor," *IBM Journal of Research and Development*, vol. 48, pp. 425–434, May 2004.

[60] Intel, "Intel Itanium Processor Firmware Specifications," http://www.intel.com/design/itanium/firmware.htm, March 2010.

[61] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. K. A. Klaiber, and J. Mattson, "The transmeta code morphing software: Using speculation, recovery, and adapative retranslation to address real-life challenges," in *Proceedings of the 1st IEEE/ACM Symposium of Code Generation and Optimization*.

[62] P. Bannon and J. Keller, "Internal architecture of alpha 21164 microprocessor," *compcon*, vol. 00, p. 79, 1995.

[63] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *CRYPTO '99: Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, pp. 388–397.

[64] C. Seger, "An introduction to formal hardware verification," University of British Columbia, Vancouver, BC, Canada, Canada, Tech. Rep., 1992.

[65] C. Kern and M. R. Greenstreet, "Formal verification in hardware design: a survey," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 4, no. 2, pp. 123–193, 1999.

[66] R. Pelanek, "Properties of state spaces and their applications," *International Journal on Software Tools and Technology Transfer*, vol. 10, no. 5, pp. 443–454, Sep. 2008.

[67] J. a. P. Marques-Silva and K. A. Sakallah, "Boolean satisfiability in electronic design automation," in *Design Automation Conference*, pp. 675–680.

[68] R. E. Bryant, "Symbolic boolean manipulation with ordered binary-decision diagrams," *Computing Surveys*, vol. 24, no. 3, pp. 293–318, September 1992.

[69] M. Srivas and M. Bickford, "Formal verification of a pipelined microprocessor," *IEEE Softw.*, vol. 7, no. 5, pp. 52–64, 1990.

[70] J. R. Burch and D. L. Dill, "Automatic verification of pipelined microprocessor control," in *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pp. 68–80.

[71] J. U. Skakkebaek, R. B. Jones, and D. L. Dill, "Formal verification of out-of-order execution using incremental flushing," in *CAV '98: Proceedings of the 10th International Conference on Computer Aided Verification*, pp. 98–109.

[72] M. N. Velev and R. E. Bryant, "Formal verification of superscale microprocessors with multicycle functional units, exception, and branch prediction," in *DAC '00: Proceedings of the 37th Annual Design Automation Conference*, pp. 112–117.

[73] V. A. Patankar, A. Jain, and R. E. Bryant, "Formal verification of an arm processor," in *Twelfth International Conference On VLSI Design*, pp. 282–287.

[74] R. Hosabettu, G. Gopalakrishnan, and M. Srivas, "Formal verification of a complex pipelined processor," *Form. Methods Syst. Des.*, vol. 23, no. 2, pp. 171–213, 2003.

[75] J. Bormann, S. Beyer, A. Maggiore, M. Siegel, S. Skalberg, T. Blackmore, and F. Bruno, "Complete formal verification of tricore2 and other processors," in *DVCon*.

[76] D. Gizopoulos, M. Psarakis, M. Hatzimihail, M. Maniatakos, A. Paschalis, A. Raghunathan, and S. Ravi, "Systematic software-based self-test for pipelined processors," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, pp. 1441–1453, September 2008.

[77] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch, "Leveraging legacy code to deploy desktop applications on the web," in *Conference on operating systems design and implementation*, pp. 339–354.

[78] A. Gluska, "Coverage-oriented verification of banias," in *Design Automation Conference*, pp. 280–285.

[79] W. Ahmed, "Implementation and verification of a cpu subsystem for multimode rf transceivers," M.S. thesis, Royal Institute of Technology, May 2010.

[80] OpenRISC.net, "Openrisc.net mailing list," http://lists.openrisc.net.

[81] OpenCores.org, "Openrisc bug tracker," http://opencores.org/openrisc,bugtracker.

[82] M. Hicks, E. Pek, and S. T. King. [Online]. Available: http://web.engr.illinois.edu/~pek1/isca2012/

[83] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, "VCC: A practical system for verifying concurrent C," in *TPHOLs 2009*, pp. 23–42.

[84] S. G. Tucker, "Microprogram control for system/360," *IBM Syst. J.*, vol. 6, pp. 222–241, December 1967.

[85] G. A. Reis, J. Chang, D. I. August, R. Cohn, and S. S. Mukherjee, "Configurable transient fault detection via dynamic binary translation," in *WORKSHOP ON ARCHITECTURAL RELIABILITY*.

[86] OpenCores.org, "Openrisc or1200 processor," http://opencores.org/or1k/OR1200_OpenRISC_Processor.

[87] TechEdSat, "Techedsat nasa program," http://ncasst.org/techedsat.html.

[88] G. Research, "Leon3 synthesizable processor," http://www.gaisler.com.

[89] OpenSPARC.net, "Oracle opensparc processor," http://opensparc.net.

[90] Digilent Inc., "Xupv5 development board," http://www.digilentinc.com/Products/Detail.cfm?NavTop=2&NavSub=599&Prod=XUPV5.

[91] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Workshop on Workload Characterization*, pp. 3–14.

[92] S. T. King, P. M. Chen, Y.-M. Wang, C. Verbowski, H. J. Wang, and J. R. Lorch, "SubVirt: Implementing malware with virtual machines," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pp. 314–327.

[93] R. Naraine, "Rutkowska faces 100% undetectable malware challenge," June 2007, ZDNet. [Online]. Available: http://www.zdnet.com/blog/security/rutkowska-faces-100-undetectable-malware-challenge/334

[94] Bob Colwell, "Personal communication," March 2009.

[95] K. Thompson, "Reflections on trusting trust," *Commun. ACM*, vol. 27, no. 8, pp. 761–763, 1984.

[96] BugTraq Mailing List, "Irssi 0.8.4 backdoor," May 2002, http://archives.neohapsis.com/archives/bugtraq/2002-05/0222.html.

[97] P. A. Karger and R. R. Schell, "Multics Security Evaluation: Vulnerability Analysis," HQ Electronic Systems Division: Hanscom AFB, Tech. Rep. ESD-TR-74-192, June 1974.

[98] P. A. Karger and R. R. Schell, "Thirty Years Later: Lessons from the Multics Security Evaluation," in *ACSAC '02: Proceedings of the 18th Annual Computer Security Applications Conference*, p. 119.

[99] CERT, "Cert advisory ca-2002-24 trojan horse openssh distribution," CERT Coordination Center, Tech. Rep., 2002, http://www.cert.org/advisories/CA-2002-24.html.

[100] CERT, "Cert advisory ca-2002-28 trojan horse sendmail distribution," CERT Coordination Center, Tech. Rep., 2002, http://www.cert.org/advisories/CA-2002-28.html.

[101] Maxtor, "Maxtor basics personal storage 3200," http://www.seagate.com/www/en-us/support/downloads/personal_storage/ps3200-sw.

[102] Apple Computer Inc., "Small number of video ipods shipped with windows virus," October 2006. [Online]. Available: http://www.apple.com/support/windowsvirus/

[103] B. Sullivan, "Digital picture frames infected with virus," January 2008, http://redtape.msnbc.com/2008/01/digital-picture.html.

[104] U. S. D. of Defense, "Mission impact of foreign influence on dod software," *Mission Impact of Foreign Influence on DoD Software*, September 2007.

[105] SPARC International Inc., "SPARC v8 processor," http://www.sparc.org.

[106] AMD, "Revision guide for amd athlon 64 and amd opteron processors," Advanced Micro Devices, Tech. Rep., August 2005.

[107] Intel, "Intel pentium 4 processor on 90nm process, specifcation update," Intel Corporation, Tech. Rep., December 2005.

[108] S. K. Sastry Hari, M.-L. Li, P. Ramachandran, B. Choi, and S. V. Adve, "mswat: low-cost hardware fault detection and diagnosis for multicore systems," in *International Symposium on Microarchitecture*, pp. 122–132.

[109] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu, "Pres: probabilistic replay with execution sketching on multiprocessors," in *Symposium on Operating Systems Principles*, pp. 177–192.